Week 6 Cheat Sheet

Statistics and Data Analysis with R

Course Link: https://www.coursera.org/learn/statistics-and-data-analysis-with-r/

Charlie Nuttelman

Here, I provide the functions in R required to perform various calculations in Week 6 of the course. The headings represent the screencasts in which you will find those concepts and examples.

Simple Linear Regression

In R, we can use the following functions to create and use a simple linear regression model ($y = \beta_0 + \beta_1 x$):

- **lm** (stats) – This function creates a linear model, the output object of which I'm giving the name **fit**.
  - In R, we can use the following to create a simple linear (straight line) regression model:
    ```
    fit <- lm(y~x)
    ```
  - Note that in the above example, the vectors **x** and **y** must have previously been defined. If **x** and **y** are found in a data frame, then we must provide as a second optional argument the data frame that contains **x** and **y** (and they must be labeled those variables in the data frame).
  - To create a model from x and y that are contained in data frame **df**, we can use:
    ```
    fit <- lm(y~x, data = df)
    ```
  - You can view the model using:
    ```
    View(fit)
    ```
  - Additional options (for example, the adjusted R-squared and standard error and P-values of the coefficients, among others) are available in the console using the summary function:
    ```
    summary(fit)
    ```
  - To store the slope and intercept as individual variables, we can use the following:
    ```
    intercept <- fit$coefficients[1]
    slope <- fit$coefficients[2]
    ```
- **predict** (stats) – Given a model fit (**fit**) created by the **lm** function, this function allows us to predict the model response (**y**) at a particular value of the independent variable (**x0**) or a vector of independent variables (**x**):
  - For a single x-value:
    ```
    predict(fit, data.frame(x=x0))
    ```
  - For example, at the single point x=25:
    ```
    predict(fit, data.frame(x=25))
    ```
  - For a vector of x-values (for example):
    ```
    x=c(15,25,35,45)
    predict(fit, data.frame(x=x))
    ```

        - OR, equivalently:
          `predict(fit, data.frame(x=c(15,25,35,45)))`

We can use the following function in R to evaluate the significance of model parameters (slope and intercept):

- **summary** (base) – Given a model fit (**fit**) created by the **lm** function, this function outputs the estimates, standard errors, t values, and P-values of the model coefficients:
  `summary(fit)`
    - The summary function also provides additional information related to the model fit, including the adjusted R-squared value.

We can use the following function in R to calculate confidence intervals on the model parameters (slope and intercept):

- **confint** (stats) – Given a model fit (**fit**) created by the **lm** function, this function outputs confidence intervals of the model coefficients:
  `confint(fit)`
    - The default confidence level is 95%, but if you wish to use a different level, we can use a second optional argument, “level=0.90” (for example, if you wish to calculate 90% confidence intervals on the model parameters).

If we wanted to calculate a confidence interval on the mean response or a prediction interval for a future observation for a simple linear regression model at an independent value **x0**, we can use the following methods in R:

- **predict** (stats) – Given a model fit (**fit**) created by the **lm** function, the **predict** function can be used to calculate a ***<u>confidence interval</u>*** on the mean response at a particular value (**x0**) or vector of values:
  ```
  predict(fit, newdata=data.frame(x=x0),
  interval='confidence')
  ```
    - For example, if **x0=34**, then we could write:
      ```
      predict(fit, newdata=data.frame(x=34),
      interval='confidence')
      ```
    - IMPORTANT: inside the **data.frame** portion, the variable (**x** in this case) must match the independent variable that we used in the **lm** function when we created the model fit (recall, we used “`fit <- lm(y~x)`” to create our model fit). If we had used “`fit <- lm(y~a)`” to create our model fit, then we would need to use “`newdata=data.frame(a=34)`” instead.
    - If we wish to calculate confidence intervals on the mean response at a vector of values, then we can use:
      ```
      predict(fit, newdata=data.frame(x=c(10,20,30,40),
      interval='confidence')
      ```
      (this is just an example)
- **predict** (stats) – Given a model fit (**fit**) created by the **lm** function, the **predict** function can be used to calculate a ***<u>prediction interval</u>*** on a future observation:
  ```
  predict(fit, newdata=data.frame(x=x0), interval='predict')
  ```

  - For example, if **x0=21**, then we could write:
    `predict(fit, newdata=data.frame(x=21), interval='predict')`
  - IMPORTANT: inside the **data.frame** portion, the variable (**x** in this case) must match the independent variable that we used in the **lm** function when we created the model fit (recall, we used "`fit <- lm(y~x)`" to create our model **fit**). If we had used "`fit <- lm(y~a)`" to create our model fit, then we would need to use "`newdata=data.frame(a=21)`" instead.

Residual Analysis

We can analyze residuals to determine model adequacy. In R, we can use the following techniques to help us with residual analysis:

- **<linear model>$residuals** – Given a model fit (**fit**) created by the **lm** function, we can output the residuals using: `fit$residuals`
- **residuals** (stats) – This function does the same exact thing as the previous bullet point. Given a model fit (fit) created by the lm function, we can output the residuals using: `residuals(fit)`
- **plot** (base) – When the argument to this function is a linear model, the **plot** function will output several residual plots. The model builder can then evaluate the model for adequacy. Please consult the screencast "Residual Analysis" (Parts 1 and 2) for more information on how to interpret the residual plots.
- We can always analyze the residuals by performing the AD test and calculating the P-value of the AD statistic for any model (using a generic model named **fit** here):
  ```
  library(nortest)
  P <- ad.test(fit$residuals)
  ```
  If the P-value is less than 0.05, the residuals are NOT normally distributed, and the model is not adequate. If the P-value is greater than 0.05, then we can assume that the residuals are normally distributed, and our model is an adequate one.

Polynomial Regression

Polynomial models are of the form $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_k x^k$ (this is referred to as a kth order polynomial). In R, we can use the following methods to create polynomial models:

- **lm** (stats) – We can use the **lm** function to create polynomial models.
  - The following creates a second order polynomial model (**y** is the dependent variable and **x** is the independent variable):
    ```
    second.order <- lm(y ~ x + I(x^2))
    ```
  - To create a 3rd order polynomial model:
    ```
    third.order <- lm(y ~ x + I(x^2) + I(x^3))
    ```
  - To create a 4th order polynomial model:
    ```
    fourth.order <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4))
    ```
  - To create a third order polynomial without the second order term:

```
mixed.order <- lm(y ~ x + I(x^3))
```

- Note that in the above examples, the vectors **x** and **y** must have previously been defined. If **x** and **y** are found in a data frame, then we must provide as a second optional argument the data frame that contains **x** and **y** (and they must be labeled those variables in the data frame).
- For example, to create a third order model from data in data frame **df**, we can use:

```
third.order <- lm(y ~ x + I(x^2) + I(x^3), data=df)
```

- **poly** (stats) – In combination with the **lm** function, this is another way to create a kth order polynomial model. Note that we must use the optional argument "**raw=TRUE**" if we wish to create typical polynomial models that do not use orthogonal terms. Note also that there is no way to create a mixed polynomial model, for example the 4th bullet point above where we have only the first and third order terms but not the second order terem.
  - To create a 2nd order polynomial using the **poly** function:

    ```
    second.order <- lm(y ~ poly(x, 2, raw=TRUE))
    ```
  - To create a 3rd order polynomial using the **poly** function:

    ```
    third.order <- lm(y ~ poly(x, 3, raw=TRUE))
    ```
  - And so on…

The **predict** function can be used to provide model predictions:

- **predict** (stats) – Given a model fit (for example, **third.order**) created by the **lm** function, this function allows us to predict the model response (**y**) at a particular value of the independent variable (**x0**) or a vector of independent variables (**x**):
  - For a single x-value:

    ```
    predict(second.order, data.frame(x=x0))
    ```
  - For example, at the single point x=5:

    ```
    predict(fit, data.frame(x=5))
    ```
  - For a vector of x-values (for example):

    ```
    x=c(100,200,300,400)
    predict(fit, data.frame(x=x))
    ```
    - OR, equivalently:

    ```
    predict(fit, data.frame(x=c(100,200,300,400)))
    ```

Similar to how we did things above with first order (straight line) models, we can use the following functions in R to analyze the adequacy of polynomial models:

- **summary** (base) – Given a model fit (for example, **third.order**) created by the **lm** function, this function outputs the estimates, standard errors, t values, and P-values of the model coefficients:

  `summary(third.order)` (for example, for the third order model)
- **confint** (stats) – Given a model fit (for example, **third.order**) created by the **lm** function, this function outputs confidence intervals of the model coefficients:

  `confint(third.order)` (for example, for the third order model)
  - The default confidence level is 95%, but if you wish to use a different level, we can use a second optional argument, "level=0.90" (for example, if you wish to calculate 90% confidence intervals on the model parameters).

- **predict** (stats) – Given a model fit (for example, **third.order**) created by the **lm** function, the **predict** function can be used to calculate a *confidence interval* on the mean response at a particular value (**x0**) or vector of values:
  ```
  predict(third.order, newdata=data.frame(x=x0),
  interval='confidence')
  ```
  - IMPORTANT: inside the **data.frame** portion, the variable (**x** in this case) must match the independent variable that we used in the **lm** function when we created the model fit.
  - If we wish to calculate confidence intervals on the mean response at a vector of values, then we can use:
    ```
    predict(third.order, newdata=data.frame(x=c(2,5,8,12),
    interval='confidence')
    ```
    (this is just an example)
- **predict** (stats) – Given a model fit (for example, **third.order**) created by the **lm** function, the **predict** function can be used to calculate a *prediction interval* on a future observation:
  ```
  predict(third.order, newdata=data.frame(x=x0),
  interval='predict')
  ```
  - IMPORTANT: inside the **data.frame** portion, the variable (**x** in this case) must match the independent variable that we used in the **lm** function when we created the model fit.

Multilinear Regression

Multilinear regression models have multiple regressor variables ($x_1, x_2, \dots, x_k$) and are of the form:

$$y = \beta_0 + \beta_1 \cdot f_1(x_1, x_2, \dots, x_k) + \beta_2 \cdot f_2(x_1, x_2, \dots, x_k) + \cdots + \beta_n \cdot f_n(x_1, x_2, \dots, x_k)$$

The simplest (and most common) multilinear model is where $f_1(x_1, x_2, \dots, x_k) = x_1$, $f_2(x_1, x_2, \dots, x_k) = x_2$, …, and $f_k(x_1, x_2, \dots, x_k) = x_k$, in which case the multilinear model becomes:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k$$

In R, we can use the following methods to create multilinear regression models:

- **lm** (stats) – We can use the **lm** function to create multilinear regression models.
  - Example #1: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$
    ```
    model1 <- lm(y ~ x1 + x2 + x3)
    ```
  - Example #2: $= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_{12} x_1 x_2 + \beta_{13} x_1 x_3 + \beta_{23} x_2 x_3$
    ```
    model2 <- lm(y~x1+x2+x3+x1*x2+x1*x3+x2*x3)
    ```
    or, equivalently:
    ```
    model2 <- lm(y~x1+x2+x3+x1:x2+x1:x3+x2:x3)
    ```
  - Note that in the above examples, the vectors **x1** through **x3** must have previously been defined. If **x1** through **x3** are found in a data frame, then we must provide as a second optional argument the data frame that contains **x1** through **x3** (and they must be labeled those variables in the data frame, and the vector **y** must also be labeled **y** in the data frame).
  - To create the Example #1 model from data in data frame **df**, we can use:
    ```
    model1 <- lm(y ~ x1 + x2 + x3, data=df)
    ```
  - To create the Example #2 model from data in a data frame **df**, we can use:

```
model2 <- lm(y~x1+x2+x3+x1*x2+x1*x3+x2*x3, data=df)
```

- If we wish to use all variables (columns) of a data frame other than **y** as regressor variables, we can simply use:

  ```
  model1 <- lm(y ~ ., data=df)
  ```

  - This creates the same model as Example #1 above
- Similarly, if we wish to create a model with all variables (columns) of a data frame other than **y** as regressor variables, but we also wish to look at binary interaction terms, then we can write:

  ```
  model2 <- lm(y~.^2, data=df)
  ```

  - This creates the same model as Example #2 above

The **predict** function can be used to provide model predictions (note that this was NOT shown in the Multilinear Regression screencast):

- **predict** (stats) – Given a model fit (for example, **model1**) created by the **lm** function, the **predict** function allows us to predict the model response (**y**) at a particular set of values of the independent variables:
  - For example, to predict the response (**y**) when x1=5, x2=350, x3=50, x4=40, and x5=250, we can use the following lines of code in R:

    ```
    new <- data.frame(x1=c(5), x2=c(350), x3=c(50),
    x4=c(40), x5=c(250))
    predict(model1, newdata=new)
    ```

  - IMPORTANT: inside the **data.frame** portion, the variables (**x1** through **x5** in this case) must match the independent variables that we used in the **lm** function when we created the model fit.

Similar to how we did things above with first order (straight line) and polynomial models, we can use the following functions in R to analyze the adequacy of polynomial models:

- **summary** (base) – Given a model fit (for example, **model1**) created by the **lm** function, this function outputs the estimates, standard errors, t values, and P-values of the model coefficients:

  `summary(model1)` (for example, for the Example #1 model above)
- **confint** (stats) – Given a model fit (for example, **model1**) created by the **lm** function, this function outputs confidence intervals of the model coefficients:

  `confint(model1)` (for example, for the Example #1 model above)
  - The default confidence level is 95%, but if you wish to use a different level, we can use a second optional argument, “level=0.90” (for example, if you wish to calculate 90% confidence intervals on the model parameters).
- **predict** (stats) – (Not shown in the Multilinear Regression screencast.) Given a model fit (for example, **model1**) created by the **lm** function, the **predict** function can be used to calculate a *<u>confidence interval</u>* on the mean response at a particular set of values of the independent variables:
  - For example, to calculate a 95% confidence interval on the mean response when x1=5, x2=350, x3=50, x4=40, and x5=250, we can use the following lines of code in R:

    ```
    new <- data.frame(x1=c(5), x2=c(350), x3=c(50),
    x4=c(40), x5=c(250))
    predict(model1, newdata=new, interval='confidence')
    ```

    - IMPORTANT: inside the **data.frame** portion, the variables (**x1** through **x5** in this case) must match the independent variables that we used in the **lm** function when we created the model fit.
- **predict** (stats) – (Not shown in the Multilinear Regression screencast.) Given a model fit (for example, **model1**) created by the **lm** function, the **predict** function can be used to calculate a *prediction interval* on a future observation at a particular set of values of the independent variables:
    - For example, to calculate a 95% prediction interval on a single future observation when x1=5, x2=350, x3=50, x4=40, and x5=250, we can use the following lines of code in R:
      ```
      new <- data.frame(x1=c(5), x2=c(350), x3=c(50),
      x4=c(40), x5=c(250))
      predict(model1, newdata=new, interval='prediction')
      ```
    - IMPORTANT: inside the **data.frame** portion, the variables (**x1** through **x5** in this case) must match the independent variables that we used in the **lm** function when we created the model fit.

Model Building Techniques

When building a polynomial or multilinear regression model from many candidate regressor variables/terms, we need to answer the following questions: 1) how many terms to include in the final model? and 2) which terms to include in the final model?

One model building technique shown in this screencast is "backward elimination", a stepwise model building strategy in which the model designer starts with a full model that contains terms that are hypothesized to contribute to the performance of the model. One at a time, the term with the largest P-value (as long as it is greater than a cutoff significance level of anywhere from 0.05 to 0.15) is removed as a candidate regressor variable and the model is re-created using the remaining terms.

To perform backward elimination in R, we can use the step function:

- **step** (stats) – Starting with a model (**model**) created using the **lm** function, the **step** function will use backward elimination to remove one term at a time until all remaining terms are less than a significance level of about 0.15:
  ```
  step(model, direction="backward")
  ```

Adjusted R-squared is typically used to evaluate the overall model performance. In R, we can store the value of this parameter using the summary function. As an example:

```
summary(fourth.order)$coefficients["I(x^3)","Estimate"]
```

Analysis of Variance

We can use the **aov** function in R to perform one-way or two-way analysis of variance (ANOVA):

- One-way ANOVA

- In one-way ANOVA, there is only a single factor (at different levels) that is hypothesized to have an effect on the output variable).
- To perform one-way ANOVA in R, we can use the **aov** function in combination with a formula and data frame (**df**):

  ```
  aov(formula, data)
  ```
- Shown here is an example using the **chickwts** data frame that's built into R. In the **chickwts** data frame, the **weights** column is a function of the type of **feed** eaten by the chicks. We can analyze the data in R using the following lines of code:

  ```
  oneway.analysis <- aov(weight ~ feed, data=chickwts)
  summary(oneway.analysis)
  ```
- As an alternative to the first line above, we can use:

  ```
  oneway.analysis <- aov(chickwts$weight ~
  chickwts$feed)
  ```
  [and use no **data** argument]

* Two-way ANOVA
  - In two-way ANOVA, also known as two-factor ANOVA, there are two factors that can be evaluated with respect to their effect on the output variable. We can also look at the interaction effect between those two input variables.
  - To perform two-way ANOVA in R, we can again use the **aov** function in combination with a formula and data frame (**df**):
  - For example, if we wish to analyze the data in the **ToothGrowth** data set, which is built into R and shows tooth length (len) data as a function of supplement type (**supp**) and **dose**, we can use the following:

    ```
    twoway.analysis <- aov(len ~ supp + dose + supp*dose,
    data=ToothGrowth)summary(twoway.analysis)
    summary(twoway.analysis)
    ```
    The first line above can also be written as:
    ```
    twoway.analysis <- aov(ToothGrowth$len ~
    ToothGrowth$supp*ToothGrowth$dose)
    ```
    We can also use the following short-hand notation to imply that we wish to analyze main effects and the interaction effect:
    ```
    twoway.analysis <- aov(len ~ supp*dose,
    data=ToothGrowth)
    summary(twoway.analysis)
    ```
  - The above will determine the main effects of supp and dose and also the interaction effect (supp:dose). If we wish to exclude the interaction effect, we can use:

    ```
    twoway.analysis <- aov(len ~ supp+dose,
    data=ToothGrowth)
    summary(twoway.analysis)
    ```