Written by Kath Fillman

PUBH6451 Coding Guide

Notes on this Guide

Items in *italics* are context-specific names (such as file names, variables, etc) that should be changed within your own code
Items in `this font and color` are code

Importing Data

Importing data into R can be done two ways:

1. If you are importing multiple documents, set the working directory using the `setwd()` to the folder that contains your files of interest.

```
setwd('C:Users/Kath/Documents/Path/To/Folder')
mydata1 <- read.csv('File1.csv')
mydata2 <- read.csv('File2.csv')
```

2. If you are importing just one document, you can import your data in a single line by writing out the path.

```
mydata <- read.csv('C:Users/Kath/Documents/Path/To/File.csv')
```

The function used to import a document depends on its data type. Most commonly data is within a CSV, so the `read.csv()` function is used. When the data is a text file `read.table()` should be chosen.
Oftentimes packages will want to be used, so you will have to load in a library. In this example, I will load-in tidyverse.

```
library(tidyverse)
```

Data Wrangling

Creating New Variables

Dichotomizing a Continuous Variable

Sometimes we want a continuous variable to be binary. Generally, this is done by choosing a cutoff and assigning everything below/above to a zero or one. Alternatively, you could choose all variables equal or not equal to a value. We will use an `ifelse` statement for this. Use any operator (`<`, `>`, `=`, `!`) or combination thereof for this. In this example the if is 1 and the else is 0.

```
mydata$new_term <- ifelse(mydata$variable > cutoff, 1, 0)
```

Creating a Multi-category Variable from a Continuous Variable

As previously stated, sometimes we want a continuous variable to be categorical- but we need more than one category. We can use `cut()` to break it into categories. The `breaks` options specifies the values to split the continuous variable (inclusive) and `labels` gives names to each new category.

```
mydata$new_term <- cut(mydata$variable, breaks=c(break1, break2, …), labels=(“A”, “B”, …))
```

Deleting Observations

This is most commonly done with missing data, so that is how we will be referring to the data we are looking to delete. To find if missing data is present, use `summary()` for continuous variables and `table()` for categorical. To remove the missing values, simply reassign the variable (or assign a new variable) with the same values using `na.omit()`.

```
mydata_without_na <- na.omit(mydata)
```

Subsetting Datasets

Sometimes we only want observations above or below a specified value. Sometimes we only want those equal to one. Use `subset()` to keep only the observations of interest. Use any operator (`<`, `>`, `=`, `!`) or combination thereof for this.

```
subset(mydata, mydata$subsetting_variable > chosen_value)
```

Multiple Linear Regression (MLR)

The basic code for multiple linear regression uses the format:

```
model <- glm(y~factor(x1)+x2+x3+..., data = mydata, family = "gaussian")
```

The `family = "gaussian"` is what specifies this as a linear model. `factor(x)` should be put around all categorical variables. You can also specify a reference group with `relevel` and `ref`.

```
model <- glm(y~relevel(factor(x1), ref=1)+x2+x3+..., data = mydata, family = "gaussian").
```

The `summ()` command is then used to determine the confidence interval (95%) and specify the number of digits after the decimal used (in this class it was 4).

```
summ(model, confint = TRUE, ci.width = 0.95, digits = 4)
```

With Interaction

If there is a binary predictor and a continuous predictor in a model, the two groups have similar slopes but different intercepts. However, it may be the case that the two groups have different intercepts AND different slopes. This is a situation of an interaction between the predictors.
First, you will fit the model (as was done above), then you will create the interaction plot by using `interact_plot()`. To signify they are interacting variables, put a `*` between them. This function comes from the `{interactions}` library.

```
interaction_model <- glm(y~factor(x1)+x2*x3, data = mydata, family = "gaussian")
interact_plot(model = interaction_model , # specify model being used
              pred = x2, # continuous predictors for x axis
              modx = x1,  # categorical predictors for the lines
              plot.points = TRUE) # plot points in addition to regression lines
```

A model should then be fitted (as above), using `summ()`.

```
summ(interaction_model, confint = TRUE, ci.width = 0.95, digits = 4)
```

Significant Interactions

If significant interactions exist, we need to obtain group means and pairwise differences. For this, we need to use the `{emmeans}` package. `emmeans()` finds the least sum square means for specified factors

```
interaction_model.stats <- emmeans(model, pairwise ~ x1*x2)
interaction_model.stats$emmeans # find group means
interaction_model.stats$contrasts |> summary(infer = TRUE) # find pairwise differences
```

Insignificant Interactions

If significant interactions do not exist, we need to obtain adjusted means and average differences. This involves creating a new model (the same as above but without the interactions denoted by a `*`). Then, we will repeat the emmeans process.

```
model <- glm(y ~ factor(x1) + factor(x2),data = mydata, family = "gaussian") # new, non-interactions model
model.stats<-emmeans(model, pairwise ~ x1 + x2)
model.stats$emmeans # find group means
model.stats.groups$contrasts |> summary(infer = TRUE) #descriptive and inferential statistics for main effects (can change contrasts to any predictor)
```

Tests for Multicategory Predictors

```
Anova(model, type = "III")
```

Model Diagnostics

Fit

Fit is often measured one of three ways: Adjusted R^2, AIC, and BIC. These values can be obtained via `summ()` from the `{jtools}` package. As a reminder, a lower AIC or BIC is preferred. Sometimes fit is measured via PRESS (Predicted Residual Sum-of-Square), which test how well the model predicts on future data. To do this, we would have to construct our own PRESS function. Note: PRESS is best used when comparing models as the output on its own doesn't mean much.

```
PRESS <- function(model) { # Create PRESS function
  i <- residuals(model)/(1 - lm.influence(model)$hat)
  sum(i^2)
  }
PRESS(model) # use function on model of choice
```

Collinearity

Collinearity happens when one predictor is highly correlated with another predictor. When there is collinearity, it can make the model unstable and give odd results. We investigate this with a correlation matrix.

```
cor(mydata[ ,c(x1, x2, x3, …)], use= "pairwise.complete.obs")
```

Multicollinearity happens when one predictor is highly correlated with some combination of two or more other predictors. Variance inflation is measured by the Variance Inflation Factor (or VIF.) The variance inflation factor ranges from 1 to infinity. A VIF below 4 indicates the variable is minimally correlated with any other predictors, a 6 VIF between 4 and 10 indicates moderate correlation, and a VIF above 10 indicates high enough correlation to affect the precision of the coefficient estimates and cast doubt on the effectiveness of the inferential results.

```
vif(model)
```

Outliers

To create a standardized residuals plot, use `plot()` and specify the third plot (`which=3`) as well as how many points you want to identify starting with the most extreme (`id.n=`). `abline()` can then be used to add a horizontal line at $\sqrt{3}$ for a visual indicator on observations worth investigating.

```
plot(model, which = 3, id.n = 10)
```

```
abline(h = sqrt(3), lty = 2)
```

Influential Observations: Cook's Distance

Measures how much the model changes when an observation is removed from the model. `influenceInxesPlot()` is in the `{cars}` package. `abline()` should once again be used to add a threshold line.

```
influenceIndexPlot(model, vars = "Cook") # plot Cook's distance
abline(h = 1, lty = 2) # add threshold line
```

Influential Observations: DFFITs

DFFITs (Difference of Fits) statistic measures how much the model changes when an observation is removed.

```
plot(dffits(model), ylab = "DFFITs") # plot DIFFITs
thresh.dff <- 2*sqrt((length(model$coefficients)-1) /
length(model$fitted.values)) # calculate threshold values
abline(h = thresh.dff, lty = 2) # add threshold line (above)
abline(h = -thresh.dff, lty = 2) # add threshold line (below)
```

Influential Observations: DFBETAs

DFBETAs (Difference of Betas) statistic measures how much a regression coefficient (betas, that is, intercept and predictor(s)) changes when an observation is removed. You can plot data for different variables by replacing `Intercept` with the variable of interest.

```
DFBETAS <- dfbetas(model) # compute DFBETAs
thresh.dfb <- 2 / sqrt(length(model$fitted.values)) # calculate threshold values
plot(DFBETAS[,"(Intercept)"], ylab="Intercept") # plot intercept
abline(h = thresh.dff, lty = 2) # add threshold line (above)
abline(h = -thresh.dff, lty = 2) # add threshold line (below)
```

Normality of Errors

Create a histogram to see normality of the residuals

```
hist(model$residuals)
```

Logistic Regression Model

Logistic regression uses the same `glm(` function as seen in MLR, but instead replaces `` `family = 'gaussian'` `` with `` `family = binomial(link="logit")` ``. We are still able to obtain the results

using `summ()`, but should also include the exp = TRUE argument to report exponentiated coefficients (i.e., odds ratios).

```
model <- glm(y ~ x1 + x2 +…, data = mydata, family =
binomial(link="logit"))
summ(model, confint = TRUE, ci.width = 0.95, digits = 4, exp = TRUE)
```

Hosmer-Lemeshow Test

Used to check for goodness-of-fit of the logistic regression model, it provides an overall or global assessment of how well the model fits the data. The null hypothesis is that the model fits the data well. We will use `hoslem.test()` from the `{ResourceSelection}` package, specifying the binary outcome variable first (must be denoted as 0/1), followed by the expected values (fitted observations from the model) for the observations

```
hoslem.test(mydata$x, fitted(model))
```

EDA for Survival Data

Kaplan-Meier Plot

Visualizing time-to-event data using the {survival} and {ggfortify} packages. To calculate overall survival time use ~ 1 and use autoplot(censor.shape="O").

```
overall_survival <- survfit(Surv(time, survival) ~ 1, data = mydata) #
calculate survival by group
autoplot(overall_survival , # plot the survival by group
     conf.int = FALSE, # do not include confidence interval
    censor.shape = "X") # change shapes of censored observations
```

To calculate survival time by group, use ~ Randomization and use autoplot(censor.shape="X").

```
survival_by_group <- survfit(Surv(time, survival) ~ Randomization, data =
mydata) # calculate survival by group
autoplot(survival_by_group, # plot the survival by group
     conf.int = FALSE, # do not include confidence interval
    censor.shape = "X") # change shapes of censored observations
```

Median Survival Time

Median survival time is when the cumulative survival drops below 50%. `survfit()` automatically estimates the median survival time and its 95% confidence interval.

Survival at a Specific Time

Use `summary()` on the `survfit` object and include `times =` containing the time points you want estimates of.

```
summary(survival_model, times=c(time1, time2, time3, …))
```

Tests for Survival Data

Log-Rank Test

Use `survdiff()` from the `{survival}` package. Effectively, use the same code as with the Kaplan-Meier plot, but change the function to `survdiff()` and add `rho = 0`. Use `~ 1` or `~Randomization` depending on if you want to use groupings.

```
survdiff(Surv(time, survival) ~ 1, data = mydata, rho = 0)
```

Wilcoxon Test

The same as above except `rho = 1`.

```
survdiff(Survtime1, time 2) ~ 1, data = mydata, rho = 1)
```

Cox Proportional Hazards Regression Model

Categorical Predictor Present

Use `coxph()` from the `{survival}` package to create the model. This is very similar to that in the Kaplan-Meier model.

```
model <- survfit(Surv(time, survival) ~ Randomization, data = mydata)
summary(model)
```

The proportional hazards assumption must then be checked; we will do this by calculating the Schoenfeld residuals and making a plot of residuals vs time.

```
model_resid <- cox.zph(model)
ggcoxzph(model1_resid)
```

Multiple Cox Proportional Hazards Regression

Use `coxph()` to create the model. Variables `x1, x2, …` are controlled for.

```
model <- coxph(Surv(time, survival) ~ Randomization + x1 + x2 + …, data = mydata)
summary(model)
```

We must then create a pairwise comparison for these predictors. This is done using `glht()` from the `{multicomp}` package. In the function the cox model is specified along with the multiple comparison method, which in this case is Tukey. We then find the confidence intervals.

```
cox.multcomp <- glht(model, linfct = mcp(m.f = "Tukey"))
exp(confint(cox.multcomp)$confint)
```

To check the proportional hazards assumption, use the same method as above.

Poisson Regression Model

Count Data with a Fixed Denominator

We will once again use `glm()`, this time with `family = poisson(link = "log")`. To display standard error values, remove the confint= and ci.width= arguments. To obtain rate ratio estimates include `exp = TRUE` in `summ()`.

```
model <- glm(y ~ x, data = mydata, family = poisson(link = "log"))
summ(model, confint = TRUE, ci.width = 0.95, digits = 4, exp = TRUE)
```

Models for Overdispersed Data

Quasi-Poisson Regression

In `glm()` specify `family = quasipoisson(link = "log")`.

```
model <- glm(y ~ x, data = mydata, family = quasipoisson(link = "log"))
```

Negative binomial regression

Use `glm.nb()` from the `{MASS}` package. The syntax is identical to the `glm()` function except for the lack of `family=` argument.

```
model <- glm.nb(y ~ x, data = mydata)
```

Binary Data with an Offset Term (Variable Denominator)

If the denominator varies, an offset term is needed. First, ensure your offset term is in a log scale.

```
model <- glm(y ~ x, data = mydata, offset = offset_varaible, family =
poisson(link = "log"))
summ(model, confint = TRUE, ci.width = 0.95, digits = 4, exp = TRUE)
```

Binary Data with an Offset Term and a Covariate

This model should look identical to the model for binary data with an offset term, except the model terms should include `+ covariate`.

```
model <- glm(y ~ x + covariate, data = mydata, offset = offset_varaible,
family = poisson(link = "log"))
```

Log-Binomial Regression Model

We will use the `glm()` function with the `family=binomial(link="log")` argument to specify log binomial regression. Use `exp=TRUE` in `summ()` if you want relative risk.

```
model <- glm(y ~ x1 +x2 + …, data = mydata, family=binomial(link="log"))
summ(model, confint = TRUE, ci.width = 0.95, digits = 4, exp = TRUE)
```

Predicted log risk can be found using the following

```
exp(predict(model, pred.data, interval="confidence"))
```

Ordinal Regression Model

Used when the outcome is an ordinal categorical variable. We will use `polr()` from the `{MASS}` package. Note that the order of your response variable, either ascending or descending, will impact the interpretation of the model results. When the factor is in ascending order, regression coefficients should be interpreted in terms of moving from one level of the response variable to the next highest level. When the factor is in descending order, regression coefficients should be interpreted in terms of moving from one level of the response variable to the next lowest level. Reverse the order of a factor's levels with the rev() function and the levels() function to suit your intended interpretation.

```
model <- polr(y ~ x1 + x2 + …, data = mydata, Hess=TRUE)
summary(model)
```

The p-value is not shown using `summary()`, so we will need a few extra lines of code to see them.

```
modelcoef <- coef(summary(model)) # extract summary table information
p.values <-  pnorm(abs(modelcoef[, "t value"]), lower.tail = FALSE) * 2 # calculates p-values based on test statistics
modelcoef.p = cbind(modelcoef, "p value" = p.values) # adds p-values as a column to the table for the null hypothesis that coef=0
round(modelcoef.p, digits=4) # rounds all numbers to four decimal places
```

The common odds ratio between groups of an explanatory variable, which is common across all levels of the ordinal response variable, can be calculated by taking the antilog of the regression coefficients. Similarly, the confidence interval limits can also be exponentiated and converted into confidence intervals for the common odds ratios.

```
exp(-coef(model)) # Common odds ratio
exp(-confint(model))  # Confidence interval for common odds
```

EDA for Model Selection

This section, unlike others, is structured in a stepwise manner; all previous sections should be worked through prior to running any code.

EDA

A correlation matrix provides multiple comparisons between variables in a matrix structure. This was also used in the collinearity section of the document. To use `cor()` specify all **numerical** variables of interest. You should also add = "complete.obs", which will remove missing values by case-wise deletion.

```
cor(mydata[ ,c(x1, x2, x3, …)], use=  "complete.obs"))
```

Fit the Models

The first step in many model selection procedures is to fit the models:

- The full model (includes all potential predictors (and all potential interaction terms)
- The null model (only includes the intercept)

Model Selection Approaches

Model selection approaches can be done using any of the model selection methods found earlier in this manual. Model selection can be done three ways: forward, backward, or stepwise. These

methods may not converge on a single, ideal model, so it is important to understand your motivations (and Ockham's razor).

Forward Selection

Forward selection builds a model by starting with a null model and sequentially adding variables into the model based on some criterion. The starting model is thus the null model. The argument, which can equal `"forward"`, `"backward"`, or `"both"`, should be `"forward"`. The `scope` argument specifies the range of models that should be considered; meaning it will consider everything between those two models.

```
forward_model <- step(nullmodel, direction = "forward", scope = list(
upper = fullmodel, lower = nullmodel))
summary(forward_model)
```

Backward Selection

Backward selection is similar to forward selection with one key difference: instead of starting with the null model and adding in variables, backward selection starts with the full model and eliminates variables. Here, we will set `direction = "backward"`.

```
backward_model <- step(nullmodel, direction = "backward", scope = list(
upper = fullmodel, lower = nullmodel))
summary(backward_model)
```

Stepwise Selection

Stepwise selection is a combination of forward and backward selection. In forward selection, once a variable is added to the model, its inclusion is never questioned. In backward selection it is the same with dropped variables. In stepwise selection, at each step, variables previously excluded from the model are considered for inclusion while variables already included in the model are considered for exclusion. You can start with either the full or null model, I've chosen to start with the null.

```
stepwise_model <- step(nullmodel, direction = "both", scope = list( upper
= fullmodel, lower = nullmodel))
summary(stepwise_model)
```