# Lab 4 Solutions

To load the data:

```
library(aod)
```

```
## Warning: package 'aod' was built under R version 3.4.2
```

```
mydata <- read.csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
#mydata <- read.csv("D:/binary.csv")
head(mydata)
```

```
##   admit gre  gpa rank
## 1     0 380 3.61    3
## 2     1 660 3.67    3
## 3     1 800 4.00    1
## 4     1 640 3.19    4
## 5     0 520 2.93    4
## 6     1 760 3.00    2
```

### 1) Get basic descriptives for the entire data set using `summary()`. View the dataset using `view()`.

```
summary(mydata)
```

```
##      admit             gre             gpa             rank
##  Min.   :0.0000   Min.   :220.0   Min.   :2.260   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:520.0   1st Qu.:3.130   1st Qu.:2.000
##  Median :0.0000   Median :580.0   Median :3.395   Median :2.000
##  Mean   :0.3175   Mean   :587.7   Mean   :3.390   Mean   :2.485
##  3rd Qu.:1.0000   3rd Qu.:660.0   3rd Qu.:3.670   3rd Qu.:3.000
##  Max.   :1.0000   Max.   :800.0   Max.   :4.000   Max.   :4.000
```

```
View(mydata)
```

### 2) How many observations are there in this dataset?

```
nrow(mydata)
```

```
## [1] 400
```

### 3) Get the standard deviations. Hint: use sapply to apply the sd function to each variable in the dataset: `sapply(mydata, sd)`. Now get the mean for the first three variables (i.e., `admit`, `gre` and `gpa`) in a similar way.

```
sapply(mydata[,-4], mean)
```

```
##    admit      gre      gpa
##   0.3175 587.7000   3.3899
```

`mydata[,-4]` indicates all the rows and all the columns except for the 4th column (the `rank`). We exclude rank as it is a categorical variable and calculating its mean is not appropriate.

**4) Convert `rank` to a factor to indicate that rank should be treated as a categorical variable.**

```
mydata$rank <- factor(mydata$rank)
```

**5) Estimate a logistic regression model using the `glm` function, and get the results using the `summary` command.**

```
glm.admit.fit <- glm(admit ~ gre + gpa + rank, data = mydata, family = "binomial")
summary(glm.admit.fit)
```

```
##
## Call:
## glm(formula = admit ~ gre + gpa + rank, family = "binomial",
##     data = mydata)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6268  -0.8662  -0.6388   1.1490   2.0790
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.989979   1.139951  -3.500 0.000465 ***
## gre          0.002264   0.001094   2.070 0.038465 *
## gpa          0.804038   0.331819   2.423 0.015388 *
## rank2       -0.675443   0.316490  -2.134 0.032829 *
## rank3       -1.340204   0.345306  -3.881 0.000104 ***
## rank4       -1.551464   0.417832  -3.713 0.000205 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 499.98  on 399  degrees of freedom
## Residual deviance: 458.52  on 394  degrees of freedom
## AIC: 470.52
##
## Number of Fisher Scoring iterations: 4
```

**6) Do you notice variable `rank` is replaced with categorical variables `rank2`, `rank3`, and `rank4` that can only take values of 0 or 1? Recall that the original variable `rank` can take values of 1, 2, 3, or 4. Why isn't a variable `rank1` needed? If rank is 1, what are the values of `rank2`, `rank3` and `rank4`?**

If `rank` is 1, then `rank2` is 0, `rank3` is 0 and `rank4` is 0.

If `rank` is 2, then `rank2` is 1, `rank3` and `rank4` are 0.

If `rank` is 3, then `rank2` is 0, `rank3` is 1 and `rank4` is 0.

If `rank` is 4, then `rank2` and `rank3` are 0 and `rank4` is 1.

**7) From the z-statistics and p-values of the variables, report which variables are statistically significant.**

The z-statistics of all the variables are large and the p-values of all the variables are small ($<0.05$). All the variables are statistically significant.

**8) Use the model to predict the training dataset and store the results to a vector of probabilities `admit.prob`.**

```
admit.prob <- predict(glm.admit.fit, type = "response")
head(admit.prob)
```

```
##         1         2         3         4         5         6
## 0.1726265 0.2921750 0.7384082 0.1783846 0.1183539 0.3699699
```

Note that this is a vector of probabilities. `admit` in the original dataset has values 0 or 1 (0 for reject and 1 for admitted).

**9) Create another vector `admit.pred` to show 0 or 1 for `admit.prob`. Let's set the value to be 0 if the probability is less than 0.5, and 1 if the probability is no less than 0.5.**

```
admit.pred <- rep(1,400)
admit.pred[admit.prob<0.5] <- 0
head(admit.pred)
```

```
## [1] 0 0 1 0 0 0
```

**10) Using `table()` function to create a confusion matrix to determines how many observations were correctly or incorrectly classified. Calculate the percentage that the observations were correctly classified.**

```
table(admit.pred, mydata$admit)
```

```
##
## admit.pred   0   1
##          0 254  97
##          1  19  30
```

```
mean(admit.pred == mydata$admit)
```

```
## [1] 0.71
```

**11) Use the model to predict the average cases in each rank, that is, four new data with mean gre, mean gpa and rank from 1 to 4.**

```
newdata1 <- with(mydata, data.frame(gre = mean(gre), gpa = mean(gpa), rank = factor(1:4)))
newdata1$admit1.prob <- predict(glm.admit.fit, newdata = newdata1, type = "response")
newdata1
```

```
##     gre    gpa rank admit1.prob
## 1 587.7 3.3899    1   0.5166016
## 2 587.7 3.3899    2   0.3522846
## 3 587.7 3.3899    3   0.2186120
## 4 587.7 3.3899    4   0.1846684
```

```
newdata1$admit1.pred <- rep(1,4)
newdata1$admit1.pred[newdata1$admit1.prob<0.5] <- 0
newdata1
```

```
##     gre    gpa rank admit1.prob admit1.pred
## 1 587.7 3.3899    1   0.5166016           1
## 2 587.7 3.3899    2   0.3522846           0
## 3 587.7 3.3899    3   0.2186120           0
```

```
## 4 587.7 3.3899    4   0.1846684           0
```

Note that the above commands add two new columns to newdata1.

Or alternatively, we can use

```
newdata2 <- data.frame(gre = mean(mydata$gre), gpa = mean(mydata$gpa), rank = factor(1:4))
newdata2$admit1.prob <- predict(glm.admit.fit, newdata = newdata2, type = "response")
newdata2
```

```
##     gre    gpa rank admit1.prob
## 1 587.7 3.3899    1   0.5166016
## 2 587.7 3.3899    2   0.3522846
## 3 587.7 3.3899    3   0.2186120
## 4 587.7 3.3899    4   0.1846684
```

```
newdata2$admit1.pred <- rep(1,4)
newdata2$admit1.pred[newdata2$admit1.prob<0.5] <- 0
```

`with(data, expression)` applies an expression to a dataset.

For more exercises and more detailed explanations, please refer to https://stats.idre.ucla.edu/r/dae/logit-regression/.

## Exercises on Functions

**1) In Session 1, we learned to combine elements into a vector using the c function, e.g. x <- c("A", "B", "C") creates a vector x with three elements. Furthermore, we can extend that vector again using c, e.g. y <- c(x, "D") creates a vector y with four elements. Write a function called fence that takes two vectors as arguments, called original and wrapper, and returns a new vector that has the wrapper vector at the beginning and end of the original.**

```
fence <- function(original, wrapper) {
  answer <- c(wrapper, original, wrapper)
  return(answer)
}

best_practice <- c("Write", "programs", "for", "people", "not", "computers")
asterisk <- "***"  # R interprets a variable with a single value as a vector
                   # with one element.
fence(best_practice, asterisk)
```

```
## [1] "***"       "Write"     "programs"  "for"       "people"    "not"
## [7] "computers" "***"
```

**2) If the variable v refers to a vector, then v[1] is the vector's first element and v[length(v)] is its last (the function length returns the number of elements in a vector). Write a function called outside that returns a vector made up of just the first and last elements of its input.**

```
outside <- function(v) {
  first <- v[1]
   last <- v[length(v)]
   answer <- c(first, last)
   return(answer)
}
```

```r
dry_principle <- c("Don't", "repeat", "yourself", "or", "others")
outside(dry_principle)
```

```
## [1] "Don't"  "others"
```

**3) Write a function that calculates number a to the power of b, but let b have a default value of 2.**

```r
powerof <- function(a, b = 2) {
  a^b
}
powerof(4)
```

```
## [1] 16
```

```r
powerof(2, 3)
```

```
## [1] 8
```

**4) Re-write the function from 3) so that a has a default value of b+1 already from the formals (from the argument definition.)**

```r
powerof <- function(a = b + 1, b = 2) {
  a^b
}
powerof()
```

```
## [1] 9
```