HW Answers

These answers are not necessarily the only answers, but they are how *I* would answer the questions.

HW1

Question 2

How many people earn more than 100,000 dollars per year *and* have `entity_county` of Los Angeles? We use `&` to combine two logical vectors:

```
cali_employees <- read.table(file = "http://tinyurl.com/pdxdgrx",
                             header = TRUE,
                             sep = "\t",
                             stringsAsFactors = FALSE)

high_pay_la_logical <- cali_employees$total_wages > 100000 &
                       cali_employees$entity_county == "Los Angeles"
selected_pays <- cali_employees$total_wages[high_pay_la_logical]
# Answer:
print(length(selected_pays))
```

```
## [1] 1278
```

Question 3

How many different position titles are held by those earning more than $100,000? We use `unique()` after extracting the positions of those earning more than 100K.

```
high_pay_logical <- cali_employees$total_wages > 100000
different_positions <- unique(cali_employees$position[high_pay_logical])
print(length(different_positions))
```

```
## [1] 84
```

Question 4

What is the position title for the highest total wage? Rather than using `max()` to get the highest total wage and comparing that answer with == (which is risky as it compares numeric values with fractional values for equality), we'll sort the data frame by `total_wage` and print the last row.

```
cali_sorted <- cali_employees[order(cali_employees$total_wages) ,]
print(cali_sorted$position[nrow(cali_sorted)])
```

```
## [1] "President, Range 0"
```

Question 5

What is the position title for the largest range between `max_classification_salary` and `min_classification_salary`? Similar to above, but we'll first add a new `range` column to order by.

```
cali_employees$range <- cali_employees$max_classification_salary - cali_employees$min_classificatio
n_salary
cali_sorted <- cali_employees[order(cali_employees$range) ,]
print(cali_sorted$position[nrow(cali_sorted)])
```

```
## [1] "Chancellor Of The California State University, Range 0"
```

Question 6

Suppose we would like to set the top 5% and bottom 5% of total wages to NA in an effort to remove "outliers" before some later data analysis. Modify the table using selective replacement to accomplish this. We'll do so using the `quantile()` function.

```
# before:
print(head(cali_employees$total_wages, n = 100))
```

```
##   [1]    805.50   8766.00   1575.00    725.00   3960.00    803.04  64305.88
##   [8]   4673.63  41509.08  79477.64    803.00 110201.18  34220.34  25788.36
##  [15]   8195.00  53923.00    700.00   3563.52  41468.80 100494.63   2478.77
##  [22]  49542.94    204.00  19285.46   6543.14    442.00  72483.32  71080.96
##  [29]  99779.18   2250.86   1652.97   2200.00  53687.28  81663.16   7551.14
##  [36]  25590.96   6758.96   4094.50   2160.90  94888.24     20.00   6159.30
##  [43]   1591.10   2923.20   8379.20  15428.34  50234.04   7212.25  16401.32
##  [50]  49627.49  50457.50  45127.92   6765.20    619.59  98883.16  95331.88
##  [57]   1154.19   5085.10   1263.60  48918.41 152077.32   1575.00    910.00
##  [64]  17281.88  61361.83 106565.32   8379.20   2428.92  37345.08   4188.41
##  [71]   1025.72  53471.52  84350.07    380.00   6909.13   4419.20  51279.84
##  [78]   7640.00   1115.00   1230.92  71964.90   4439.25    582.35   3657.60
##  [85]  53459.98    290.00    749.02   9886.40  52751.38    748.75   2487.60
##  [92]   2067.20   6489.63  11991.36  32264.54  51587.28   2704.50   9382.20
##  [99]  90997.24  43604.00
```

```
percentiles <- quantile(cali_employees$total_wages, c(0.05, 0.95))
cali_employees$total_wages[cali_employees$total_wages <= percentiles[1]] <- NA
cali_employees$total_wages[cali_employees$total_wages >= percentiles[2]] <- NA
# after:
print(head(cali_employees$total_wages, n = 100))
```

```
##   [1]   805.50  8766.00  1575.00   725.00  3960.00   803.04 64305.88
##   [8]  4673.63 41509.08 79477.64   803.00       NA 34220.34 25788.36
##  [15]  8195.00 53923.00   700.00  3563.52 41468.80       NA  2478.77
##  [22] 49542.94       NA 19285.46  6543.14   442.00 72483.32 71080.96
##  [29]       NA  2250.86  1652.97  2200.00 53687.28 81663.16  7551.14
##  [36] 25590.96  6758.96  4094.50  2160.90 94888.24       NA  6159.30
##  [43]  1591.10  2923.20  8379.20 15428.34 50234.04  7212.25 16401.32
##  [50] 49627.49 50457.50 45127.92  6765.20   619.59       NA       NA
##  [57]  1154.19  5085.10  1263.60 48918.41       NA  1575.00   910.00
##  [64] 17281.88 61361.83       NA  8379.20  2428.92 37345.08  4188.41
##  [71]  1025.72 53471.52 84350.07   380.00  6909.13  4419.20 51279.84
##  [78]  7640.00  1115.00  1230.92 71964.90  4439.25   582.35  3657.60
##  [85] 53459.98   290.00   749.02  9886.40 52751.38   748.75  2487.60
##  [92]  2067.20  6489.63 11991.36 32264.54 51587.28  2704.50  9382.20
##  [99] 90997.24 43604.00
```

HW2

We just want to write a function called `paired_test()` that either runs a t.test or a wilcox.test, depending in the result of shapiro.test (using wilcox if shapiro says that the data are likely to be non-normal by reporting a p-value less than 0.1). The result should be a single-row data frame, with a pvalue column and a “test” column indicating either “ttest_mean” or “wilcox_median”. The function’s input should be a two-column data frame to run the tests on.

```r
paired_test <- function(df) {
  x <- df[[1]]
  y <- df[[2]]
  diff <- x - y
  shapiro_res <- shapiro.test(diff)
  if(shapiro_res$p.value < 0.1) {
    test <- t.test(x, y, paired = TRUE)
    res <- data.frame(pvalue = test$p.value, test = "ttest_mean")
    return(res)
  } else {
    test <- wilcox.test(x, y, paired = TRUE)
    res <- data.frame(pvalue = test$p.value, test = "wilcox_median")
    return(res)
  }
}

# a data frame with two normally-distributed columns
df_norm <- data.frame(col1 = rnorm(100, mean = 5, sd = 2),
                      col2 = rnorm(100, mean = 10, sd = 2))
res_norm <- paired_test(df_norm)
print(res_norm)
```

```
##         pvalue          test
## 1 5.349152e-18 wilcox_median
```

```
# a data frame with non-normally-distributed columns
df_nonnorm <- data.frame(firstcol = rexp(100, rate = 2.5),
                         secondcol = rexp(100, rate = 1.3))
res_nonnorm <- paired_test(df_nonnorm)
print(res_nonnorm)
```

```
##         pvalue       test
## 1 0.0004405782 ttest_mean
```

HW 3

Question 1

We want to describe the differences between these three lines:

```
expr <- expr[expr$id %in% keep_ids, ]
expr <- expr[expr$id == keep_ids, ]
expr <- expr[keep_ids, ]`
```

1: All three lines use `df[row_selector, col_selector]` syntax. In this case, the row selector is a logical vector produced by the `%in%` operator, which produces a logical vector of length `expr$id` (left hand side) indicating which elements of `expr$id` match any element in `keep_ids` (the right hand side). This uses logical indexing to extract just the rows wanted.

2: This line also uses logical indexing, but this time `==` is used instead of `%in`. Since `==`, like most operators, works on an element-by-element basis, the result will be incorrect: some of the wanted rows will be extracted but not all. The reason is that the first element `expr$id` will be compared to the first of `keep_ids`, then the second element of each will be compared, and so on, until we run out of elements in the shorter vector (`keep_ids` in this case). At that point, the shorter `keep_ids` will be recycled. Thus, rows will only be kept where `expr$id` happens to coincide with elements of `keep_ids` through this recycling process.

3. Because the row selector in this case is a character vector, R will attempt to extract rows by *row name*. However, the row names are by default “1”, “2”, “3”, and so on, not entries from the `id` column. Even if we had said `rownames(expr) <- expr$id`, this solution would still not work because duplicate row names are not allowed but the `id` column is non-unique.

In the following lines:

```
expr <- subset(expr, id %in% keep_ids)
expr <- subset(expr, id == keep_ids)
expr <- subset(expr, keep_ids)`
```

The first two lines are interpreted as identical to the first two lines above and produce exactly the same results. The third line results in an error, as subset() requires a logical selection vector.

Question 2

We want to write a function called `numeric_only()` that, given a data frame, returns a copy data frame with all non-numeric columns removed. We'll use the fact that data frames are a type of list, using `lapply()` with `is.numeric()` to produce a logical list of column to keep (which we'll then convert to a vector with `unlist()`), before using `df[]` to extract the columns of interest using list-based logical selection. Many concepts in a short bit of code!

```
numeric_only <- function(df) {
  keep_log_list <- lapply(df, is.numeric)
  keep_log_vec <- unlist(keep_log_list)
  answer <- df[keep_log_vec]
  return(answer)
}

df1 <- data.frame(id = c("PRQ", "XL2", "BB4"), val = c(23, 45.6, 62))
df2 <- data.frame(srn = c(4461, 5144), name = c("Mel", "Ben"), age = c(27, 24))

res_1 <- numeric_only(df1)
res_2 <- numeric_only(df2)
print(res_1)
```

```
##    val
## 1 23.0
## 2 45.6
## 3 62.0
```

```
print(res_2)
```

```
##    srn age
## 1 4461  27
## 2 5144  24
```

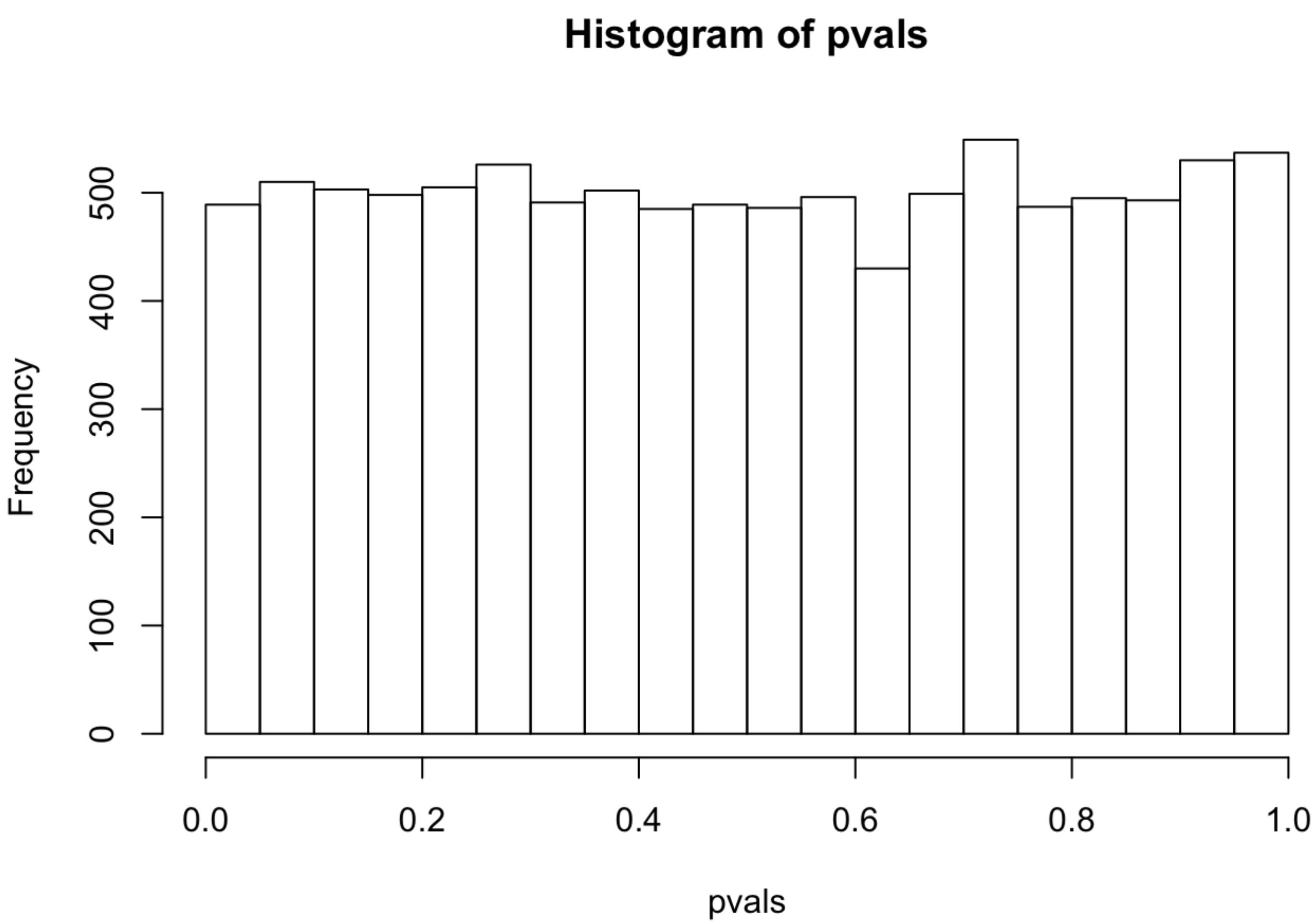
Question 3

Here we'll write a function that takes a parameter `x`, ignores it, as well as two parameters with defaults `mean1 = 0.0` and `mean2 = 0.0`, and runs a t.test on two 100-element normal samples. It returns the pvalue.

```
ttest_pval <- function(x, mean1 = 0.0, mean2 = 0.0) {
  samp1 <- rnorm(100, mean = mean1, sd = 1)
  samp2 <- rnorm(100, mean = mean2, sd = 1)
  tres <- t.test(samp1, samp2)
  return(tres$p.value)
}
```

And the code we are given to run:

```
alist <- as.list(seq(1:10000))
pvals <- lapply(alist, ttest_pval, mean1 = 0.0, mean2 = 0.0)
pvals <- unlist(pvals)
hist(pvals)
```



Questions: What does this test reveal? What is the above code doing, and why does it work? Why does the function need to take a parameter x that isn't even used?

This test reveals that T-tests, when comparing many samples drawn from the same distribution, result in p-values that are evenly distributed between 0 and 1. This is expected, as is falls in line with the definition of a p-value. The code above calls `ttest_pval()` once for each element of a given list; in this case it is called for each element of a list of 10000 numbers. The elements of the list, which are given as `x` in the function call, are ignored: they are simply used as a method to get `lapply()` to run the same function 10000 times with identical parameters (amongst the parameters that are actually used).

When we change `mean1 = 0.1` in the `lapply()`, we see that the pvalue are biased towards 0.0 as more tests are statistically significant:

```
alist <- as.list(seq(1:10000))
pvals <- lapply(alist, ttest_pval, mean1 = 0.1, mean2 = 0.0)
pvals <- unlist(pvals)
hist(pvals)
```