# ETC3555 2018 - Lab 1

Introduction to data munging with tidyverse

*Cameron Roach*

*25 July, 2018*

## Preliminaries

This lab will focus on working with data. We will cover loading, munging, summarising and fitting models to a dataset. Our dataset will contain quality ratings of wine samples and several predictor variables.

The most important package we will be using is `dplyr`. It contains several functions (sometimes referred to as verbs) for manipulating data. Some of the most common verbs and their purpose are

- `select` selects columns
- `mutate` adds new variables
- `filter` filters based on a condition
- `arrange` sorts based on column values
- `group_by` groups data
- `summarise` summarise all values within a group.

A handy reference is Hadley Wickham's R for Data Science available at r4ds.had.co.nz. Excellent cheat sheets can be downloaded from www.rstudio.com/resources/cheatsheets/.

## Exercises

### Exercise 1: Loading data

1. Download the red and white wine quality datasets from the UCI Machine Learning Repository (archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/).
2. Look at these files in a text editor.
3. Check the documentation for `read_csv`.
4. Select an appropriate function and delimiter to read these files into two separate data frames.

```r
library(tidyverse)

red_df <- read_delim('data/winequality-red.csv', delim = ";")
white_df <- read_delim('data/winequality-white.csv', delim = ";")
```

The `readr` functions store data in tibbles. Tibbles are an updated type of data frame and have several advantages over base `R` data frames. One such feature is preserving spaces in the column names. Unfortunately, this can cause issues with other `R` packages (e.g. `randomForest`) so we will convert the white space to underscores. Run the following code to remove all white space from column names.

```r
names(red_df) <- str_replace_all(names(red_df), " ", "_")
names(white_df) <- str_replace_all(names(red_df), " ", "_")
```

### Exercise 2: Adding variables

We can use `dplyr`'s `mutate` function to add variables to our data frames.

1. Add a column to your red and white wine data frames indicating the wine colour.

2. Bind these data frames into a single data frame using the `bind_rows` function.
3. Add a Boolean variable that is TRUE if the quality is greater than or equal to 7 (hint: use the `if_else` and `mutate` functions).
4. Add a unique identifier for each wine sample based on row number (hint: use `row_number`). This will be important when converting the table from long to wide formats.

```
red_df <- mutate(red_df, wine = "Red")
white_df <- mutate(white_df, wine = "White")
wine_df <- bind_rows(red_df, white_df)
wine_df <- mutate(wine_df, good_quality = if_else(quality >= 7, TRUE, FALSE))
wine_df <- mutate(wine_df, id = row_number())
wine_df <- select(wine_df, id, wine, everything())
wine_df <- na.omit(wine_df)
```

**Exercise 3: Pipes**

The pipe operator, `%>%`, takes the output from a previous operation and adds it as the first argument in the next expression. In other words

```
mutate(red_df, wine = "Red")
```

is equivalent to

```
red_df %>% mutate(wine = "Red")
```

This is very useful when carrying out many operations on a data frame. Chaining verbs together leads to easy to read code. Try incorporating pipes into your previous solution. Can you further tidy the code by combining `mutate` functions?

```
wine_df <- red_df %>%
  bind_rows(white_df) %>%
  mutate(good_quality = if_else(quality >= 7, TRUE, FALSE),
         id = row_number()) %>%
  select(id, wine, everything())
```

What if we are interested in more than two categories? Say we wish to try and predict if a wine is good, bad or average. We can do this by creating a data frame with our class definitions and then joining it with our original data frame.

1. Create a new data frame with a quality column (numbers 3, 4, … 9) and quality category column (Bad, Average, Good).
2. Join this new "quality" data frame with your original wine data frame.

```
quality_df <- tribble(
  ~ quality, ~ quality_cat,
  3, "Bad",
  4, "Bad",
  5, "Average",
  6, "Average",
  7, "Good",
  8, "Good",
  9, "Good"
)

wine_df <- wine_df %>%
  inner_join(quality_df, by = "quality")
```

### Exercise 4: Filtering and arranging data

We can arrange the data frame and filter based on conditions using the `arrange` and `filter` functions. Filter for red wines only and arrange in decreasing wine quality.

```
wine_df %>%
  filter(wine == "Red") %>%
  arrange(desc(quality))
```

```
## # A tibble: 1,599 x 16
##       id  wine fixed_acidity volatile_acidity citric_acid residual_sugar
##    <int> <chr>         <dbl>            <dbl>       <dbl>          <dbl>
##  1   268   Red           7.9             0.35        0.46            3.6
##  2   279   Red          10.3             0.32        0.45            6.4
##  3   391   Red           5.6             0.85        0.05            1.4
##  4   441   Red          12.6             0.31        0.72            2.2
##  5   456   Red          11.3             0.62        0.67            5.2
##  6   482   Red           9.4             0.30        0.56            2.8
##  7   496   Red          10.7             0.35        0.53            2.6
##  8   499   Red          10.7             0.35        0.53            2.6
##  9   589   Red           5.0             0.42        0.24            2.0
## 10   829   Red           7.8             0.57        0.09            2.3
## # ... with 1,589 more rows, and 10 more variables: chlorides <dbl>,
## #   free_sulfur_dioxide <dbl>, total_sulfur_dioxide <dbl>, density <dbl>,
## #   pH <dbl>, sulphates <dbl>, alcohol <dbl>, quality <dbl>,
## #   good_quality <lgl>, quality_cat <chr>
```

We can also check for missing values. Run the following code to see which columns have `NA` values. Don't worry about understanding what's going on here - we'll come back to this in a later exercise.

```
wine_df %>%
  map(~ sum(is.na(.x))) %>%
  unlist()
```

```
##                   id                 wine        fixed_acidity
##                    0                    0                    0
##     volatile_acidity          citric_acid       residual_sugar
##                    0                    0                    0
##            chlorides  free_sulfur_dioxide total_sulfur_dioxide
##                    0                    0                    2
##              density                   pH            sulphates
##                    0                    0                    0
##              alcohol              quality         good_quality
##                    0                    0                    0
##          quality_cat
##                    0
```

We see that there are two `NA` values in the `total_sulfur_dioxide` column. Use the filter function to remove these rows (hint: use the `is.na` function and `!` logical operator in your condition).

```
wine_df <- wine_df %>%
  filter(!is.na(total_sulfur_dioxide))

wine_df %>%
  map(~ sum(is.na(.x))) %>%
  unlist()
```

```
##                   id                 wine       fixed_acidity 
##                    0                    0                    0 
##     volatile_acidity          citric_acid       residual_sugar 
##                    0                    0                    0 
##            chlorides  free_sulfur_dioxide total_sulfur_dioxide 
##                    0                    0                    0 
##              density                   pH            sulphates 
##                    0                    0                    0 
##              alcohol              quality         good_quality 
##                    0                    0                    0 
##          quality_cat 
##                    0
```

### Exercise 5: Reshaping data

Our dataset is currently in a wide table format. Variables are spread out across the top of our data frame. Convert the data frame into a long table format using the `gather` function from the `tidyr` package.

```
wine_df_lng <- wine_df %>%
  gather(variable, value, -c(id, wine, quality_cat, good_quality))

wine_df_lng
```

```
## # A tibble: 77,940 x 6
##       id  wine good_quality quality_cat      variable value
##    <int> <chr>        <lgl>       <chr>         <chr> <dbl>
##  1     1   Red        FALSE     Average fixed_acidity   7.4
##  2     2   Red        FALSE     Average fixed_acidity   7.8
##  3     3   Red        FALSE     Average fixed_acidity   7.8
##  4     4   Red        FALSE     Average fixed_acidity  11.2
##  5     5   Red        FALSE     Average fixed_acidity   7.4
##  6     6   Red        FALSE     Average fixed_acidity   7.4
##  7     7   Red        FALSE     Average fixed_acidity   7.9
##  8     8   Red         TRUE        Good fixed_acidity   7.3
##  9     9   Red         TRUE        Good fixed_acidity   7.8
## 10    10   Red        FALSE     Average fixed_acidity   7.5
## # ... with 77,930 more rows
```

You can undo this using the `spread` function to convert the data frame back to a wide format. Spend some time experimenting with reshaping the data frame.

```
spread(wine_df_lng, variable, value)
```

```
## # A tibble: 6,495 x 16
##       id  wine good_quality quality_cat alcohol chlorides citric_acid
##    <int> <chr>        <lgl>       <chr>   <dbl>     <dbl>       <dbl>
##  1     1   Red        FALSE     Average     9.4     0.076        0.00
##  2     2   Red        FALSE     Average     9.8     0.098        0.00
##  3     3   Red        FALSE     Average     9.8     0.092        0.04
##  4     4   Red        FALSE     Average     9.8     0.075        0.56
##  5     5   Red        FALSE     Average     9.4     0.076        0.00
##  6     6   Red        FALSE     Average     9.4     0.075        0.00
##  7     7   Red        FALSE     Average     9.4     0.069        0.06
##  8     8   Red         TRUE        Good    10.0     0.065        0.00
##  9     9   Red         TRUE        Good     9.5     0.073        0.02
## 10    10   Red        FALSE     Average    10.5     0.071         0.36
## # ... with 6,485 more rows, and 9 more variables: density <dbl>,
## #   fixed_acidity <dbl>, free_sulfur_dioxide <dbl>, pH <dbl>,
## #   quality <dbl>, residual_sugar <dbl>, sulphates <dbl>,
## #   total_sulfur_dioxide <dbl>, volatile_acidity <dbl>
```

### Exercise 6: Summarising data

Now that we have our data in a long table format we can easily calculate statistics for each variable. Use `dplyr`'s `group_by` function to group the data frame by wine colour, variable and quality category. Calculate the median and standard deviation of each variable using the `summarise` function. The `summarise` function reduces all the values in each group to a single value.

```
wine_df_stats <- wine_df_lng %>%
  group_by(wine, quality_cat, variable) %>%
  summarise(median = median(value),
            sd = sd(value))

wine_df_stats
```

```
## # A tibble: 72 x 5
## # Groups:   wine, quality_cat [?]
##     wine quality_cat            variable  median           sd
##    <chr>       <chr>               <chr>   <dbl>        <dbl>
##  1   Red     Average             alcohol 10.0000  0.972649051
##  2   Red     Average           chlorides  0.0800  0.047573496
##  3   Red     Average         citric_acid  0.2400  0.187867413
##  4   Red     Average             density  0.9968  0.001815838
##  5   Red     Average       fixed_acidity  7.8000  1.682976556
##  6   Red     Average free_sulfur_dioxide 14.0000 10.413017132
##  7   Red     Average                  pH  3.3100  0.152398433
##  8   Red     Average             quality  5.0000  0.499947492
##  9   Red     Average      residual_sugar  2.2000  1.398942091
## 10   Red     Average           sulphates  0.6100  0.167284035
## # ... with 62 more rows
```

Create a wide table with variables as the key and median as values. Do you notice any obvious differences between good and bad samples?

```
wine_df_stats %>%
  select(-sd) %>%
  spread(variable, median)
```

```
## # A tibble: 6 x 14
## # Groups:   wine, quality_cat [6]
##    wine quality_cat alcohol chlorides citric_acid density fixed_acidity
##   <chr>       <chr>   <dbl>     <dbl>       <dbl>   <dbl>         <dbl>
## 1   Red     Average    10.0     0.080        0.24 0.99680           7.8
## 2   Red         Bad    10.0     0.080        0.08 0.99660           7.5
## 3   Red        Good    11.6     0.073        0.40 0.99572           8.7
## 4 White     Average    10.0     0.044        0.32 0.99438           6.8
## 5 White         Bad    10.1     0.046        0.30 0.99410           6.9
## 6 White        Good    11.5     0.037        0.31 0.99173           6.7
## # ... with 7 more variables: free_sulfur_dioxide <dbl>, pH <dbl>,
## #   quality <dbl>, residual_sugar <dbl>, sulphates <dbl>,
## #   total_sulfur_dioxide <dbl>, volatile_acidity <dbl>
```

*Note: this would be better done by visualising density functions for each variable. We will look at this in the next lab.*

### Exercise 7: Fitting multiple models

Let's fit some models! We will fit separate models to the red and white wine datasets to predict if a wine is good, average or bad. To do this (in a tidy way) we first need to look at the `purrr` package. This package is useful when we want to work with lists. Two important functions we will use are `nest` and `map`. Look at the documentation for these two functions and try to understand what each does.

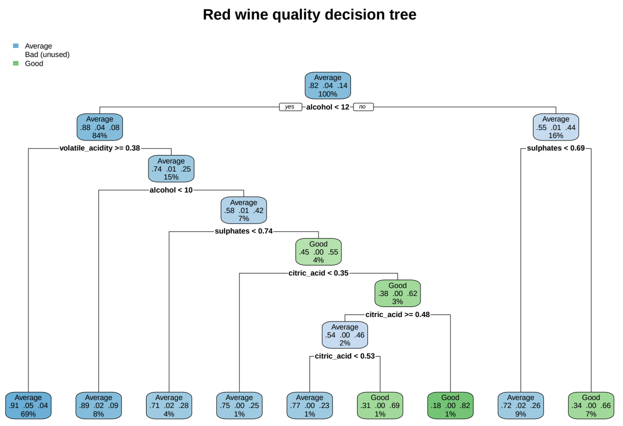
Now, create a data frame nested by wine type. Then use `map` to fit a decision tree to each. As a starting point, your decision tree should predict the wine quality category based on alcohol, volatile_acidity, citric_acid and sulphates. You'll need to ensure you have the `rpart` library installed and loaded. Once you have successfully fit the decision tree, try adding another column with a random forest fit to the same data.

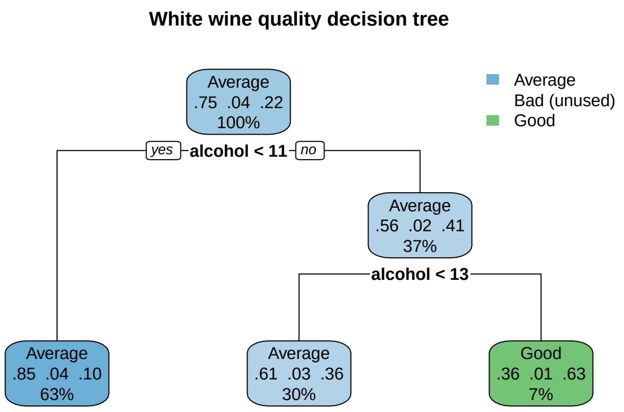
Hint: here you are trying to add a list column, and so you will `mutate` on your nested data column. However, since a nested column is essentially just a list, we now need to apply our model fitting function to each element within that list (i.e. both the red wine and white wine datasets). Hence, we will need to use `map` within the `mutate` function.

```r
library(rpart)
library(randomForest)

fit_dt <- function(x) {
  x <- as_data_frame(x)  # for resample objects

  rpart(quality_cat ~ alcohol + volatile_acidity + citric_acid + sulphates,
        data = x)
}

fit_rf <- function(x) {
  x <- as_data_frame(x) %>%  # for resample objects
    mutate(quality_cat = factor(quality_cat, ordered = FALSE))  # randomForest requires factor response

  randomForest(quality_cat ~ alcohol + volatile_acidity + citric_acid + sulphates,
               data = x)
}

fit_df <- wine_df %>%
  group_by(wine) %>%
  nest() %>%
  mutate(model_dt = map(data, fit_dt),
         model_rf = map(data, fit_rf))
```

Apply the following code to your data frame to view the decision trees for red and white wine quality. What does the `walk2` function do? How does it differ to `map`? Is there a `map` equivalent?

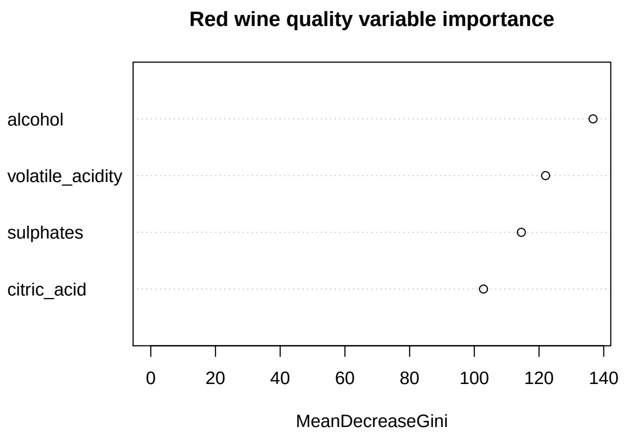
```r
library(rpart.plot)

plot_dt <- function(tree, title) {
  rpart.plot(tree, main = title)
}

walk2(fit_df$model_dt, paste(fit_df$wine, "wine quality decision tree"), plot_dt)
```

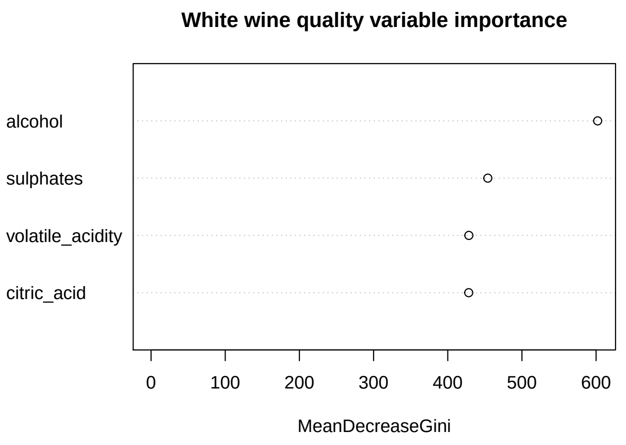




Adapt the code above to produce variable importance plots for the two random forest models (hint: use the `varImpPlot` function).

```r
plot_var_imp <- function(rf, title) {
  varImpPlot(rf, main = title)
}

walk2(fit_df$model_rf, paste(fit_df$wine, "wine quality variable importance"), plot_var_imp)
```





The random forest package automatically creates a confusion matrix when fit. See if you can print the confusion matrices for the red wine and white wine models. You can use the `map` function again.

```r
map(fit_df$model_rf, "confusion")
```

```
## [[1]]
##           Average Bad Good class.error
## Average     1256   8   53  0.04631739
## Bad           57   5    1  0.92063492
## Good          90   0  127  0.41474654
##
## [[2]]
##           Average Bad Good class.error
## Average     3428  18  209   0.0621067
## Bad          142  31   10   0.8306011
## Good         389   3  668   0.3698113
```

**Exercise 8: Cross-validation**

Look at the `modelr` package located at github.com/tidyverse/modelr. Apply cross validation and fit separate models to each training set. Pick a suitable performance metric and compare your decision tree and random forest classifiers. Which performs best on the validation sets? How does the per-class classification error compare with the confusion matrix generated above?

Can you spot a flaw in our approach to model validation?

```
library(modelr)

# Should have created an out of sample data set and only
# done CV on the in sample dataset.
wine_cv_df <- wine_df %>%
  group_by(wine) %>%
  nest() %>%
  mutate(folds = map(data, crossv_kfold, id = "id_cv", k = 3)) %>%
  unnest(folds) %>%  # can .drop = FALSE to stop data column being dropped
  mutate(fit_train_dt = map(train, fit_dt),
         fit_train_rf = map(train, fit_rf),
         pred_test_dt = map2(fit_train_dt, test, predict, type = "class"),
         pred_test_rf = map2(fit_train_rf, test, predict),
         actual_test = map(test, ~ as_data_frame(.x)$quality_cat))

# Work out per class accuracy
wine_cv_df %>%
  unnest(pred_test_dt, pred_test_rf, actual_test) %>%
  select(wine, DT = pred_test_dt, RF = pred_test_rf, actual = actual_test) %>%
  gather(model, pred, DT, RF) %>%
  mutate(correct = pred == actual) %>%
  group_by(wine, model, actual) %>%
  summarise(classification_error = (n() - sum(correct))/n()) %>%
  spread(model, classification_error)
```

```
## # A tibble: 6 x 4
## # Groups:   wine [2]
##    wine  actual         DT         RF
##   <chr>   <chr>      <dbl>      <dbl>
## 1   Red Average 0.04403948 0.04479879
## 2   Red     Bad 0.98412698 0.90476190
## 3   Red    Good 0.69124424 0.45622120
## 4 White Average 0.04459644 0.06593707
## 5 White     Bad 1.00000000 0.89071038
## 6 White    Good 0.75943396 0.42641509
```

**Exercise 9: Grid search**

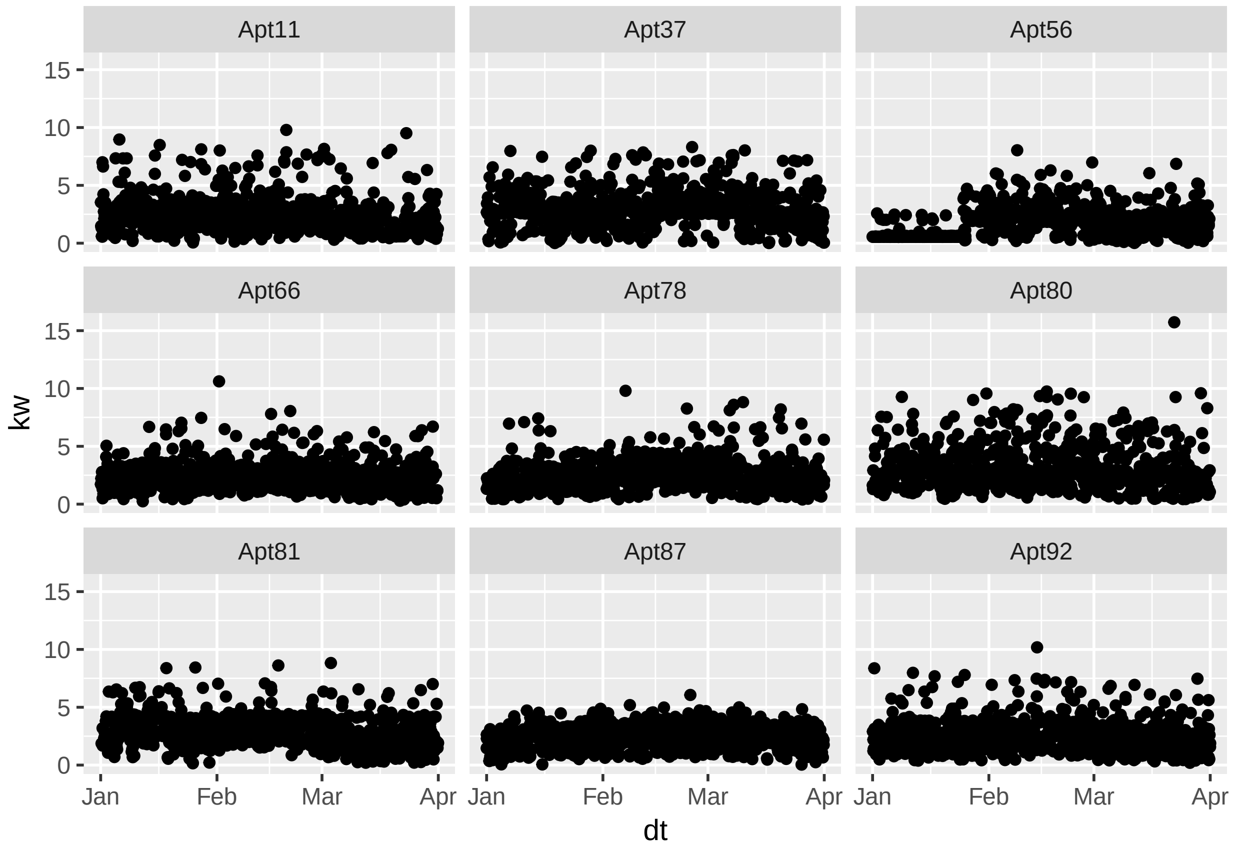
Now that you understand the fundamentals of working with data, fitting models and carrying out model validation we can look at an even nicer way of approaching the problem. Furthermore - we can also incorporate grid search into our analysis! Look through the documentation for the pipelearner package located at github.com/drsimonj/pipelearner. Work through the examples in the README of this package. Plot a learning curve.

**Exercise 10: A real world example**

The UMass Smart dataset hosts electricity usage and weather data sets. Download the apartment electricity usage dataset at traces.cs.umass.edu/index.php/Smart/Smart. This dataset contains electricity consumption data from 114 single family apartments between 2014 and 2016.

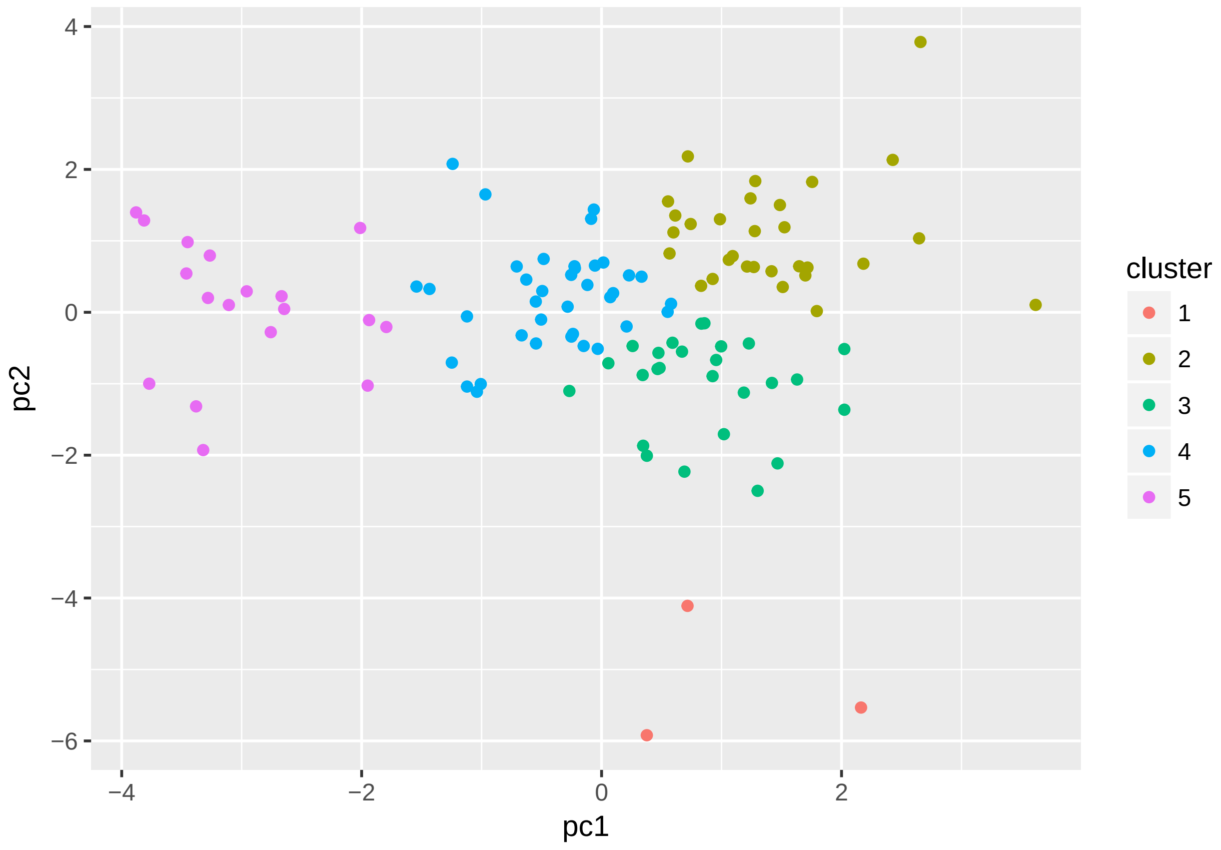
Load the 2015 data into your `R` session and filter for the months January, February and March (hint: the `lubridate` package has several useful functions for working with dates and time). Can you find different classes of energy usage behaviour? For example, do some families use significantly more electricity in the evenings? Do others use more in mornings? How might we describe these classes?

```r
library(tidyverse)
library(lubridate)
library(moments)

# Load data
read_and_id_csv <- function(file) {
  read_csv(file, col_names = c("dt", "kw")) %>%
    mutate(building = str_extract(file, "Apt[0-9]+"),
           file = file) %>%
    select(building, everything())
}

trace_df <- list.files(path = "data/apartment/2015",
                       pattern = ".csv",
                       full.names = TRUE) %>%
  map(read_and_id_csv) %>%
  bind_rows() %>%
  select(-file) %>%  # only used to check regex correct
  filter(month(dt) %in% c(1, 2, 3))

# Check some of the data to make sure things look ok
trace_df %>%
  filter(building %in% sample(building, 9)) %>%
  group_by(building) %>%
  sample_frac(0.1) %>%
  ggplot(aes(x = dt, y = kw)) +
  geom_point() +
  facet_wrap(~building)
```

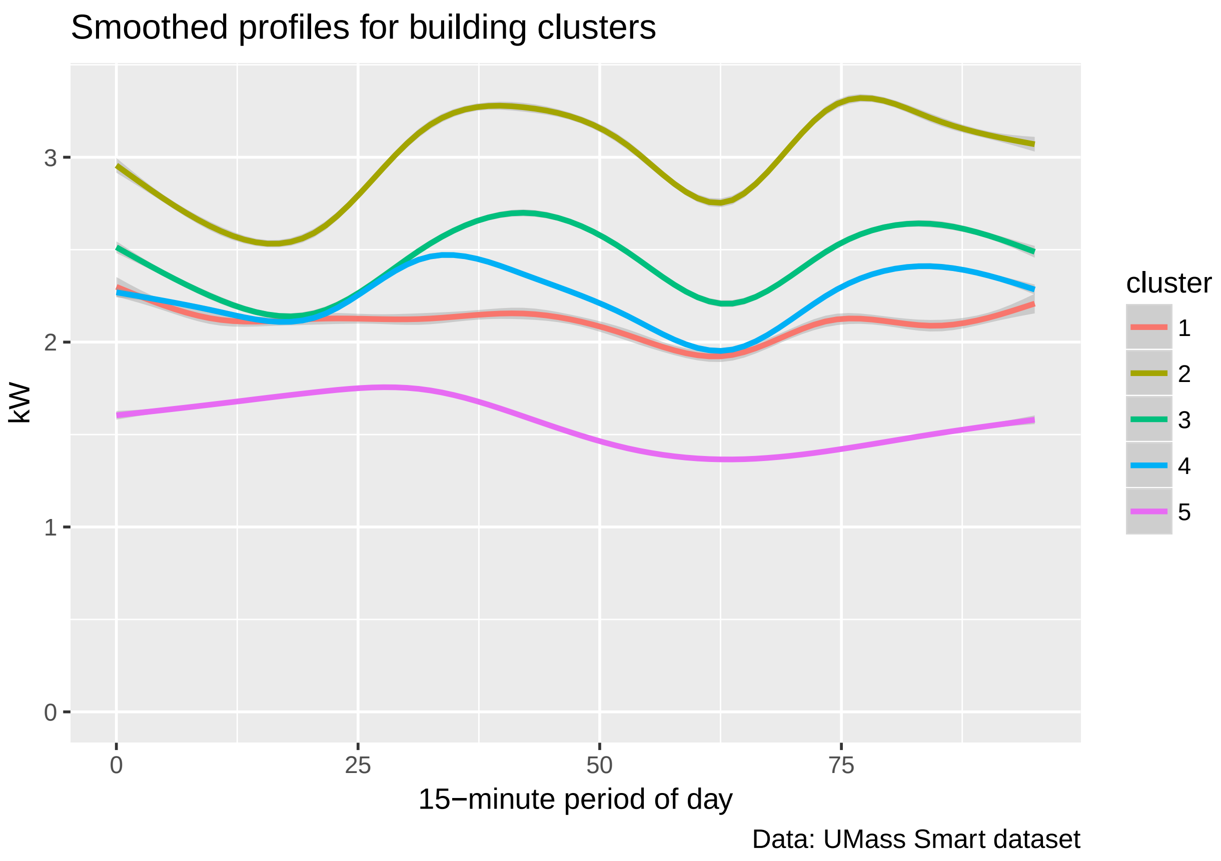


```r
# Calculate features for our energy re
#
# Note that this is just one way of approaching the problem - you may
# well think of better features to improve separation!
trace_stats_df <- trace_df %>%
  select(-dt) %>%
  group_by(building) %>%
  summarise(mean = mean(kw),
            sd = sd(kw),
            skew = skewness(kw),
            kurtosis = kurtosis(kw),
            max = max(kw),
            min = min(kw))

# Apply pca
trace_pca <- prcomp(scale(trace_stats_df[, -1]))

# K-means on first two PCA components
trace_kmean <- kmeans(trace_pca$x[,1:2], 5)

# Add PCA and clustering values into trace stats data frame
cluster_df <- data_frame(
  building = trace_stats_df$building,
  pc1 = trace_pca$x[,1],
  pc2 = trace_pca$x[,2],
  cluster = factor(trace_kmean$cluster)
)
```

```r
# Plot clusters on principal components
ggplot(cluster_df, aes(x = pc1, y = pc2, colour = cluster)) +
  geom_point()
```



```r
# Plot smoothed daily profiles
cluster_df %>%
  select(building, cluster) %>%
  inner_join(trace_df, by = "building") %>%
  mutate(period = hour(dt)*4 + minute(dt)/15) %>%
  ggplot(aes(x = period, y = kw, colour = cluster)) +
  geom_smooth() +
  ylim(0, NA) +
  labs(title = "Smoothed profiles for building clusters",
       caption = "Data: UMass Smart dataset",
       x = "15-minute period of day",
       y = "kW")
```



```
# It's interesting to look at the actual time series for these clusters. I'll
# leave that as an exercise...
```