Feature engineering

Task: To predict students' eventual course outcomes based on the data about their interaction with online course activities during the first few weeks of the course

Rationale: To be able to timely warn students who are at the risk of having low course outcome

Data set: The data used below is an adapted (simplified) version of the publicly available Moodle Learning Analytics data set

Load the required R packages (additional ones will be loaded as needed)

```r
library(ggplot2)
library(dplyr)

data_dir = "data"
```

Exploratory data analysis

Load the data

We have two data sets, one with logged events data and the other with the student grades

```r
events = read.csv(paste(data_dir, "events.csv", sep='/'))
grades = read.csv(paste(data_dir, "grades.csv", sep = '/'))
```

```r
glimpse(events)
```

```
## Rows: 95,626
## Columns: 3
## $ user   <chr> "9d744e5bf", "91489f7a9", "278a75edf", "53d6ab60c", "aab7ad69f"~
## $ action <chr> "Assignment", "Assignment", "Assignment", "Assignment", "Assign~
## $ ts     <chr> "2019-10-26 09:37:12", "2019-10-26 09:09:34", "2019-10-18 12:05~
```

```r
glimpse(grades)
```

```
## Rows: 130
## Columns: 2
## $ user  <chr> "6eba3ff82", "05b604102", "111422ee7", "b4658c3a9", "e6ec47f29",~
## $ grade <dbl> 2.626970, 4.670169, 9.244600, 0.000000, 8.238179, 5.604203, 5.09~
```

Note that based on the data set as originally given, we cannot make predictions about student course outcomes: the only variables we have are `action` and `ts` and these - in the given form - cannot be used for any meaningful prediction. So, we have to create ('engineer') new features that will be used for the prediction task.

Explore logged events

We will first focus on the time stamps (datetime) data and explore how we can make use of it

```
events$ts[1:10]
```

```
##  [1] "2019-10-26 09:37:12" "2019-10-26 09:09:34" "2019-10-18 12:05:28"
##  [4] "2019-10-19 13:28:37" "2019-10-15 23:38:13" "2019-10-18 17:51:43"
##  [7] "2019-10-18 15:22:56" "2019-10-22 13:46:51" "2019-10-15 14:58:17"
## [10] "2019-10-19 13:28:38"
```

To handle datetime data easily and effectively, we will use the `lubridate` R package

```
library(lubridate)
```

Let's start by transforming time stamp data, from char format, to the R's format suitable for working with date and time

```
?parse_date_time
```

```
ev1 = parse_date_time(events$ts[1], "ymd HMS")
ev1
```

```
## [1] "2019-10-26 09:37:12 UTC"
```

```
class(ev1)
```

```
## [1] "POSIXct" "POSIXt"
```

Note that `POSIXct` is the native R's format for representing date and time and it stores datetime data as the number of seconds from January 1st, 1970, the so-called Unix timestamp.

```
ev2 = parse_date_time(events$ts[2], "ymd HMS")
ev1 - ev2
```

```
## Time difference of 27.63333 mins
```

```
class(ev1 - ev2)
```

```
## [1] "difftime"
```

An alternative is to use functions specialised for some commonly used datetime formats - for example, for the above one, we could have used the `ymd_hms()` function:

```
ymd_hms(events$ts[1])
```

```
## [1] "2019-10-26 09:37:12 UTC"
```

Now, apply it to the entire column with the timestamp data:

```r
events$dt = parse_date_time(events$ts, "ymd HMS")
```

```r
events[1:10, c("ts","dt")]
```

```
##                     ts                  dt
## 1  2019-10-26 09:37:12 2019-10-26 09:37:12
## 2  2019-10-26 09:09:34 2019-10-26 09:09:34
## 3  2019-10-18 12:05:28 2019-10-18 12:05:28
## 4  2019-10-19 13:28:37 2019-10-19 13:28:37
## 5  2019-10-15 23:38:13 2019-10-15 23:38:13
## 6  2019-10-18 17:51:43 2019-10-18 17:51:43
## 7  2019-10-18 15:22:56 2019-10-18 15:22:56
## 8  2019-10-22 13:46:51 2019-10-22 13:46:51
## 9  2019-10-15 14:58:17 2019-10-15 14:58:17
## 10 2019-10-19 13:28:38 2019-10-19 13:28:38
```

Remove the original (character) timestamp variable, as it is no longer needed

```r
events$ts = NULL
```

When we have datetime (timestamp) data in the R's native datetime format, we can use functions from the *lubridate* package to easily extract individual pieces of date and time:

```r
date(events$dt) |> head() |> print()
```

```
## [1] "2019-10-26" "2019-10-26" "2019-10-18" "2019-10-19" "2019-10-15"
## [6] "2019-10-18"
```

```r
# day(events$dt) |> head() |> print()
# hour(events$dt) |> head() |> print()
# isoweek(events$dt) |> head() |> print()
```

For more functions, see *lubridate* cheatsheets available, for example, here

We can also order the events, for each user, based on their time stamp

```r
# note: arrange f. comes from dplyr package and allows for sorting a data frame based on one or more co
events |> arrange(user, dt) -> events

head(events)
```

```
##       user       action                  dt
## 1 00a05cc62  Course_view 2019-09-09 18:18:01
## 2 00a05cc62  Course_view 2019-09-09 18:19:02
## 3 00a05cc62 Instructions 2019-09-09 18:23:03
## 4 00a05cc62  Course_view 2019-09-09 18:23:04
## 5 00a05cc62  Course_view 2019-09-09 20:30:10
## 6 00a05cc62  Course_view 2019-09-09 20:31:11
```

Let's now examine the time range the data is available for. This should (roughly) coincide with the start and the end of the course

```r
course_start <- min(events$dt)
print(course_start)
```

```
## [1] "2019-09-09 14:08:01 UTC"
```

```r
course_end <- max(events$dt)
print(course_end)
```

```
## [1] "2019-10-27 19:27:41 UTC"
```

We can also easily compute the course length (in weeks):

```r
# ?difftime

course_len = difftime(course_end, course_start, units="week")
```

Alternatively:

```r
course_len = course_end - course_start
# in this case, the 'auto' option is applied to choose the most suitable unit option
# see the documentation of `difftime` for details

# course_len = course_len / 7
```

Since we want to make predictions based on the first couple of weeks data, we need to add a variable denoting the course week when an event occurred - this will allow us to subset the data set and take only events that took place in the given number of course weeks.

```r
get_course_week <- function(dt) {
  week = isoweek(dt)  # this will give us, for each timestamp, the week of the year
  week - min(week) + 1 # this will give us the course week, as a number between 1 and 7
}

events$course_week = get_course_week(events$dt)
```

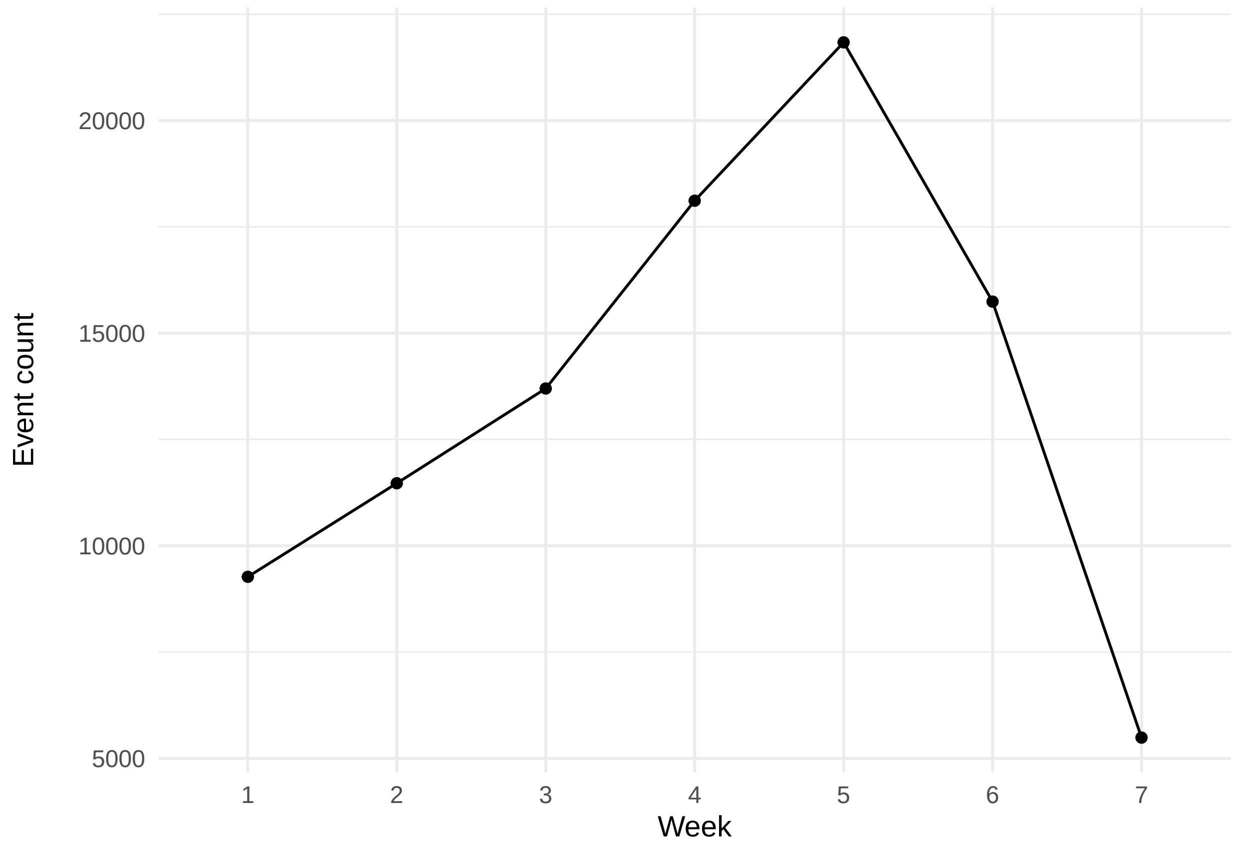
Check the distribution of event counts across the course weeks

```r
table(events$course_week)
```

```
##
##     1     2     3     4     5     6     7
##  9271 11471 13699 18115 21838 15742  5490
```

We can also do that visually (distribution is always better understood when preented visually)

```r
table(events$course_week) |> as.data.frame() -> events_per_week

ggplot(events_per_week, aes(x = Var1, y = Freq, group=1)) +
  geom_line() + geom_point() +
  labs(x = "Week", y = "Event count\n") +
  theme_minimal()
```



Let’s now examine the character variable that represents different types of learning-related actions

```
table(events$action)|> prop.table() |> round(digits = 3)
```

```
##
## Applications   Assignment  Course_view       Ethics     Feedback      General
##        0.008        0.077        0.264        0.011        0.033        0.035
##   Group_work Instructions     La_types   Practicals       Social       Theory
##        0.342        0.068        0.020        0.105        0.023        0.014
```

Some of these actions refer to individual course topics, that is, to the access to lecture materials on distinct course topics. These are: General, Applications, Theory, Ethics, Feedback, La_types. We will mark them all as “Lecture”, both to reduce the level of granularity, and to avoid mixture of activity types and activity topics

```
course_topics <- c("General", "Applications", "Theory",  "Ethics", "Feedback", "La_types")

events$action = ifelse(test = events$action %in% course_topics,
                      yes = "Lecture",
                      no = events$action)
```

```
table(events$action)|> prop.table() |> round(digits = 3)
```

```
##
##   Assignment  Course_view   Group_work Instructions      Lecture   Practicals
##        0.077        0.264        0.342        0.068        0.121        0.105
##       Social
##        0.023
```

Before moving to the feature engineering, let's also prepare the outcome variable

Examine grades data

Examine the summary statistics and distribution of the final grade

```
summary(grades$grade)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   5.666   7.954   7.254   9.006  10.000
```

We'll add *course_outcome* as a binary variable indicating if a student had a good or weak course outcome. Students whose final grade is above 50th percentile (median) will be considered as having good course outcome (HIGH), the rest will be considered as having weak course outcome (LOW)

```
grades$course_outcome = ifelse(test=grades$grade > median(grades$grade), yes = "High", no = "Low")
grades$course_outcome = as.factor(grades$course_outcome)
```

```
table(grades$course_outcome)
```

```
##
## High  Low
##   65   65
```

This gives us a perfectly balanced data set for the outcome prediction (classification) task.

As done before, we remove the variable we used for creating the outcome variable

```
grades$grade <- NULL
```

Save pre-processed data

```
saveRDS(grades, paste(data_dir, "grades_preprocessed.RDS", sep='/'))
saveRDS(events, paste(data_dir, "events_preprocessed.RDS", sep='/'))
```

Feature creation (engineering)

If pre-processing not done, load the pre-processed data

```
# grades = readRDS(paste(data_dir, "grades_preprocessed.RDS", sep='/'))
# events = readRDS(paste(data_dir, "events_preprocessed.RDS", sep='/'))
```

We will create the following features for each student:

- Number of active days, where active days are days with at least one learning action

- Average number of actions per day (considering active days only)
- Average time distance between two consecutive active days
- Total number of each type of learning actions

Since the intention is to create a prediction model based on the first few weeks of the logged course data - say, **first three weeks** - we will start by creating a subset of the `events` data that includes only logged actions from the first three course weeks:

```
events_subset = events[events$course_week <= 3, ]
```

```
dim(events_subset)
```

```
## [1] 34441     4
```

(1) Compute the number of active days (= days with at least one learning action)

To compute the number of active days, we need to add the date variable

```
events_subset$active_day = date(events_subset$dt)

events_subset[sample(1:nrow(events_subset), 10), c("dt", "active_day")]
```

```
##                        dt active_day
## 66407 2019-09-27 09:39:55 2019-09-27
## 16092 2019-09-14 10:26:52 2019-09-14
## 29717 2019-09-13 22:02:00 2019-09-13
## 1411  2019-09-28 18:44:40 2019-09-28
## 25595 2019-09-29 14:49:58 2019-09-29
## 34789 2019-09-21 08:58:50 2019-09-21
## 15386 2019-09-27 11:09:22 2019-09-27
## 67583 2019-09-28 08:27:41 2019-09-28
## 49092 2019-09-23 09:16:35 2019-09-23
## 36100 2019-09-16 16:51:00 2019-09-16
```

Now, let's compute the number of active days per student

```
events_subset |>
  group_by(user) |>
  summarise(adays_cnt = n_distinct(active_day)) -> active_days_count
```

```
head(active_days_count)
```

```
## # A tibble: 6 x 2
##   user      adays_cnt
##   <chr>         <int>
## 1 00a05cc62         7
## 2 042b07ba1         7
## 3 046c35846         3
## 4 05b604102         3
## 5 0604ff3d3         4
## 6 077584d71        17
```

```r
summary(active_days_count$adays_cnt)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    3.00    6.00   10.00   10.55   14.00   20.00
```

(2) Next, compute average number of actions per active day

First, we'll compute the number of actions per user per (active) day

```r
events_subset |> count(user, active_day) -> cnt_per_user_aday

head(cnt_per_user_aday, 10)
```

```
##         user active_day  n
## 1  00a05cc62 2019-09-09 10
## 2  00a05cc62 2019-09-11 13
## 3  00a05cc62 2019-09-14  6
## 4  00a05cc62 2019-09-15 14
## 5  00a05cc62 2019-09-20 35
## 6  00a05cc62 2019-09-27  7
## 7  00a05cc62 2019-09-28  9
## 8  042b07ba1 2019-09-12  9
## 9  042b07ba1 2019-09-13 70
## 10 042b07ba1 2019-09-17 47
```

Then, we'll average the number of action (n) for each user

```r
cnt_per_user_aday |>
  group_by(user) |>
  summarise(avg_action_cnt = median(n)) -> avg_cnt_per_aday
```

```r
head(avg_cnt_per_aday)
```

```
## # A tibble: 6 x 2
##   user      avg_action_cnt
##   <chr>              <dbl>
## 1 00a05cc62             10
## 2 042b07ba1             57
## 3 046c35846             19
## 4 05b604102             19
## 5 0604ff3d3             29
## 6 077584d71             11
```

(3) Average time distance between two consecutive active days

We will do this in a step-wise manner to make the computation clearer

First, reduce the data set to distinct active days for each user. The rationale: since we need to compute time distance between consecutive active days, we need just one occurrence of each active day

```
events_subset |>
  distinct(user, active_day) |>
  arrange(user, active_day) -> distinct_user_aday
```

```
head(distinct_user_aday, 10)
```

```
##         user active_day
## 1  00a05cc62 2019-09-09
## 2  00a05cc62 2019-09-11
## 3  00a05cc62 2019-09-14
## 4  00a05cc62 2019-09-15
## 5  00a05cc62 2019-09-20
## 6  00a05cc62 2019-09-27
## 7  00a05cc62 2019-09-28
## 8  042b07ba1 2019-09-12
## 9  042b07ba1 2019-09-13
## 10 042b07ba1 2019-09-17
```

Next, add a variable that will for each active day store the immediate previous active day (if it exists) Note the use of two new functions: i) `mutate` allows for creating new variables in a data frame; ii) `lag` returns the value of the given variable lagged for the given number of steps, by default 1

```
distinct_user_aday |>
  group_by(user) |>
  mutate(prev_active_day = lag(active_day)) -> distinct_user_aday

head(distinct_user_aday, 10)
```

```
## # A tibble: 10 x 3
## # Groups:   user [2]
##    user      active_day prev_active_day
##    <chr>     <date>     <date>
##  1 00a05cc62 2019-09-09 NA
##  2 00a05cc62 2019-09-11 2019-09-09
##  3 00a05cc62 2019-09-14 2019-09-11
##  4 00a05cc62 2019-09-15 2019-09-14
##  5 00a05cc62 2019-09-20 2019-09-15
##  6 00a05cc62 2019-09-27 2019-09-20
##  7 00a05cc62 2019-09-28 2019-09-27
##  8 042b07ba1 2019-09-12 NA
##  9 042b07ba1 2019-09-13 2019-09-12
## 10 042b07ba1 2019-09-17 2019-09-13
```

Now we can add a variable for the time difference between an active day and the immediate previous (active) day. We compute this for each user separately.

```
distinct_user_aday |>
  group_by(user) |>
  mutate(aday_diff = ifelse(is.na(prev_active_day),
                          yes = NA,
                          no = difftime(active_day, prev_active_day, units = "days"))) -> distinct_us
```

```r
head(distinct_user_aday, 10)
```

```
## # A tibble: 10 x 4
## # Groups:   user [2]
##    user      active_day prev_active_day aday_diff
##    <chr>     <date>     <date>              <dbl>
##  1 00a05cc62 2019-09-09 NA                     NA
##  2 00a05cc62 2019-09-11 2019-09-09              2
##  3 00a05cc62 2019-09-14 2019-09-11              3
##  4 00a05cc62 2019-09-15 2019-09-14              1
##  5 00a05cc62 2019-09-20 2019-09-15              5
##  6 00a05cc62 2019-09-27 2019-09-20              7
##  7 00a05cc62 2019-09-28 2019-09-27              1
##  8 042b07ba1 2019-09-12 NA                     NA
##  9 042b07ba1 2019-09-13 2019-09-12              1
## 10 042b07ba1 2019-09-17 2019-09-13              4
```

Finally, we can compute, for each user, the average (median) time distance between any two consecutive active days

```r
distinct_user_aday |>
  group_by(user) |>
  summarise(avg_aday_diff = median(aday_diff, na.rm=T)) -> avg_aday_time_diff
```

```r
head(avg_aday_time_diff, 10)
```

```
## # A tibble: 10 x 2
##    user      avg_aday_diff
##    <chr>             <dbl>
##  1 00a05cc62           2.5
##  2 042b07ba1           2.5
##  3 046c35846           8
##  4 05b604102           8
##  5 0604ff3d3           3
##  6 077584d71           1
##  7 081b100cf           2
##  8 0857b3d8e           1
##  9 0ec99ce96           1.5
## 10 0ef305578           1
```

```r
summary(avg_aday_time_diff$avg_aday_diff)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   1.000   1.250   1.945   2.500   8.000
```

(4) Total number of each type of learning action

We can easily compute, for each student, the number of actions of distinct types:

```r
events_subset |>
  count(user, action) -> counts_per_user
```

```r
head(counts_per_user)
```

```
##        user       action  n
## 1 00a05cc62  Course_view 36
## 2 00a05cc62   Group_work 18
## 3 00a05cc62 Instructions  9
## 4 00a05cc62      Lecture 11
## 5 00a05cc62   Practicals 16
## 6 00a05cc62       Social  4
```

However, this format of the computed action counts cannot serve as the input for a classification algorithm. What we need is a data frame with columns corresponding to distinct types of actions (e.g., Course_view, Group_work, Lecture, …) and values of those columns being counts of students' actions of the given type. In particular, we want unique values of the `action` column to be the columns in a data frame, whereas values of the column `n` should be used to populate those columns.

This is, in fact, a very frequent problem and a solution is readily available from the the *tidyr* package. It is known as transforming data from the *long format* (the one given above) to the *wide format* (our target, see the output of the cell below). The name *long format* refers to (relatively) small number of columns and (relatively) large number of rows. On the other hand, the name *wide format* refers to the opposite - (relatively) large number of columns and (relatively) small number of rows.

```r
library(tidyr)
```

The function we'll use is `pivot_wider`

```r
?pivot_wider
```

```r
counts_per_user |>
  pivot_wider(id_cols = user,
              names_from = action,
              values_from = n,
              values_fill = 0) -> counts_per_user_wide
```

```r
head(counts_per_user_wide)
```

```
## # A tibble: 6 x 8
##   user  Course_view Group_work Instructions Lecture Practicals Social Assignment
##   <chr>       <int>      <int>        <int>   <int>      <int>  <int>      <int>
## 1 00a0~          36         18            9      11         16      4          0
## 2 042b~         113         93           36      15         32     26          5
## 3 046c~          22          1            6      32         19      0          1
## 4 05b6~          20          1            7      31         18      0          4
## 5 0604~          32         40            4       3         23      0          0
## 6 0775~         107         98           34      48         46     14          3
```

Compare the dimensions of the long and wide formats (of the same data):

```r
print(paste("Long format:", nrow(counts_per_user), "rows and", ncol(counts_per_user), "columns"))
```

```
## [1] "Long format: 844 rows and 3 columns"
```

```r
print(paste("Wide format:", nrow(counts_per_user_wide), "rows and", ncol(counts_per_user_wide), "column
```

```
## [1] "Wide format: 128 rows and 8 columns"
```

Now that we have computed all the features, we should integrate them into one data frame. This can be easily done using the `inner_join` f. from *dplyr* package. This function works like the inner join operation in data bases.

```r
active_days_count |>
  inner_join(avg_cnt_per_aday) |>
  inner_join(avg_aday_time_diff) |>
  inner_join(counts_per_user_wide) -> feature_set
```

```
## Joining with ‘by = join_by(user)‘
## Joining with ‘by = join_by(user)‘
## Joining with ‘by = join_by(user)‘
```

Note that here we didn't specify the column(s) to be used for joining the data frames and the `inner_join` function used the column that the two data frames it was joining had in common (in this case the `user` column)

```r
head(feature_set, 10)
```

```
## # A tibble: 10 x 11
##    user      adays_cnt avg_action_cnt avg_aday_diff Course_view Group_work
##    <chr>         <int>          <dbl>         <dbl>       <int>      <int>
##  1 00a05cc62         7           10             2.5          36         18
##  2 042b07ba1         7           57             2.5         113         93
##  3 046c35846         3           19             8            22          1
##  4 05b604102         3           19             8            20          1
##  5 0604ff3d3         4           29             3            32         40
##  6 077584d71        17           11             1           107         98
##  7 081b100cf        10           27.5           2            81         88
##  8 0857b3d8e        17           29             1           126        187
##  9 0ec99ce96         9           11             1.5          74         36
## 10 0ef305578        13           13             1           107        140
## # i 5 more variables: Instructions <int>, Lecture <int>, Practicals <int>,
## #   Social <int>, Assignment <int>
```

Add the outcome variable

```r
feature_set |>
  inner_join(grades) -> feature_set
```

```
## Joining with ‘by = join_by(user)‘
```

Check the structure and completeness of the resulting data set

```r
glimpse(feature_set)
```

```
## Rows: 128
## Columns: 12
## $ user           <chr> "00a05cc62", "042b07ba1", "046c35846", "05b604102", "06~
## $ adays_cnt      <int> 7, 7, 3, 3, 4, 17, 10, 17, 9, 13, 3, 13, 7, 7, 5, 6, 3,~
## $ avg_action_cnt <dbl> 10.0, 57.0, 19.0, 19.0, 29.0, 11.0, 27.5, 29.0, 11.0, 1~
## $ avg_aday_diff  <dbl> 2.5, 2.5, 8.0, 8.0, 3.0, 1.0, 2.0, 1.0, 1.5, 1.0, 7.0, ~
## $ Course_view    <int> 36, 113, 22, 20, 32, 107, 81, 126, 74, 107, 45, 75, 68,~
## $ Group_work     <int> 18, 93, 1, 1, 40, 98, 88, 187, 36, 140, 25, 115, 58, 19~
## $ Instructions   <int> 9, 36, 6, 7, 4, 34, 34, 32, 18, 33, 18, 49, 18, 9, 16, ~
## $ Lecture        <int> 11, 15, 32, 31, 3, 48, 9, 56, 27, 37, 9, 34, 14, 12, 48~
## $ Practicals     <int> 16, 32, 19, 18, 23, 46, 37, 29, 26, 58, 3, 27, 5, 16, 1~
## $ Social         <int> 4, 26, 0, 0, 0, 14, 4, 38, 2, 16, 8, 13, 19, 3, 3, 7, 0~
## $ Assignment     <int> 0, 5, 1, 4, 0, 3, 3, 9, 1, 24, 1, 17, 11, 0, 2, 16, 1, ~
## $ course_outcome <fct> Low, Low, Low, Low, Low, High, High, High, Low, High, H~
```

```r
all(complete.cases(feature_set))
```

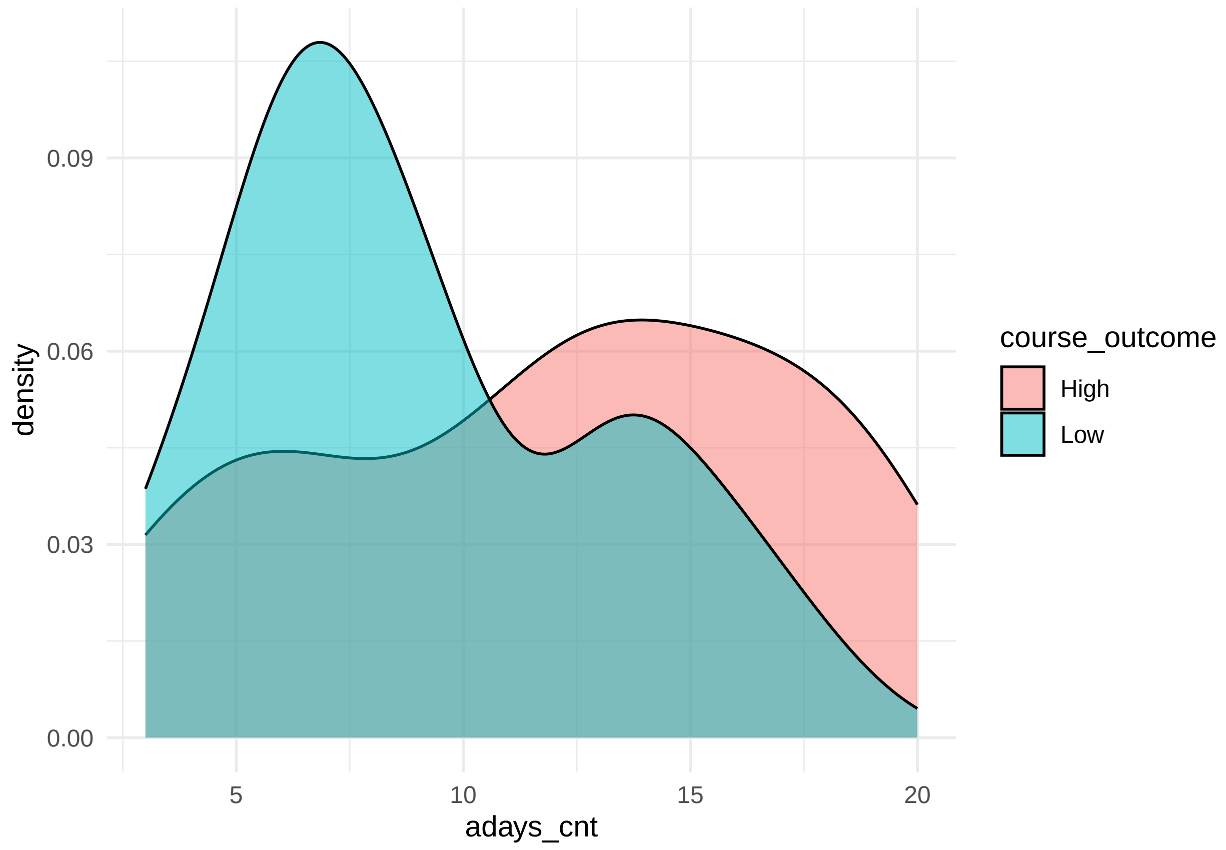
```
## [1] TRUE
```

Examine feature relevance

Examine the relevance of features for the prediction of the outcome variable

Let's first recall how we can do it for one continuous variable

```r
ggplot(feature_set,
       aes(x = adays_cnt, fill=course_outcome)) +
  geom_density(alpha=0.5) +
  theme_minimal()
```

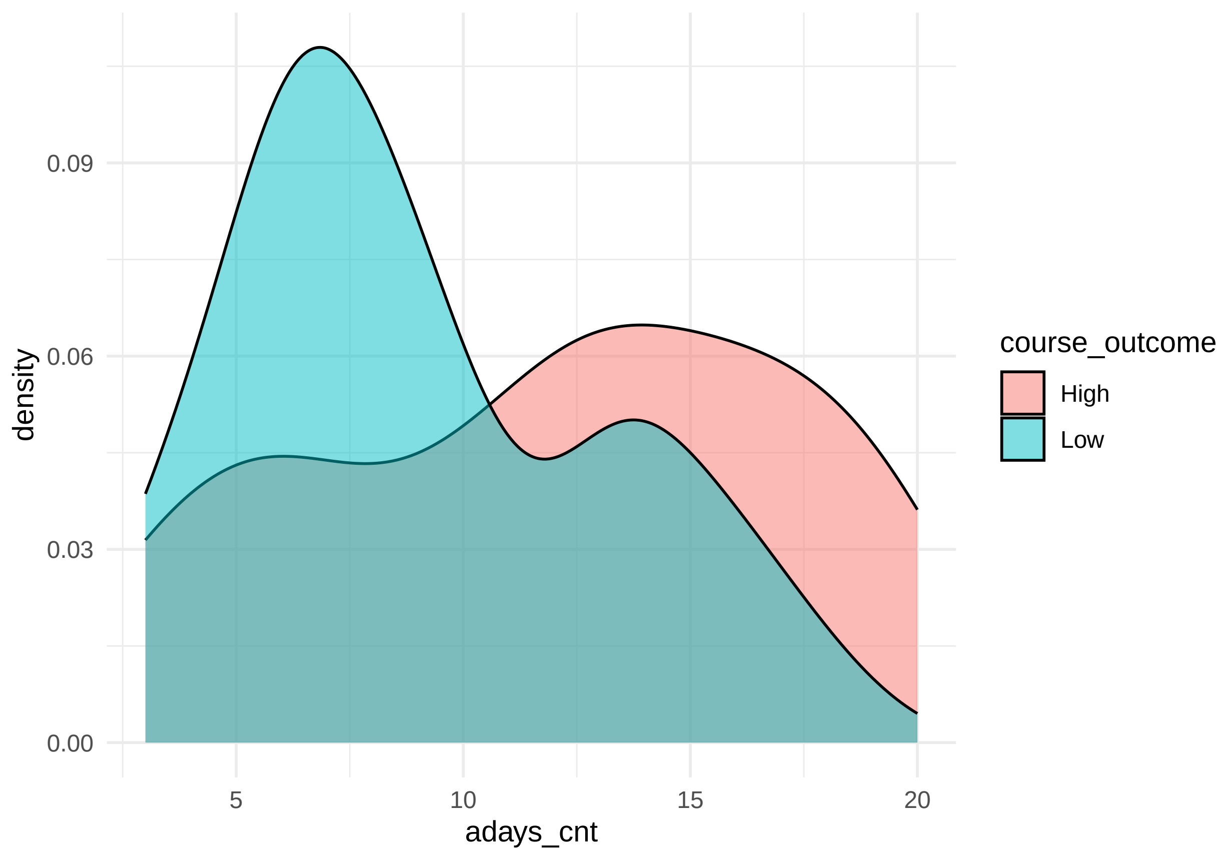


Clearly, the number of active days is relevant for predicting the course outcome, since the plot indicates that students with more active days tend to have high course performance

Now, do for all features at once:

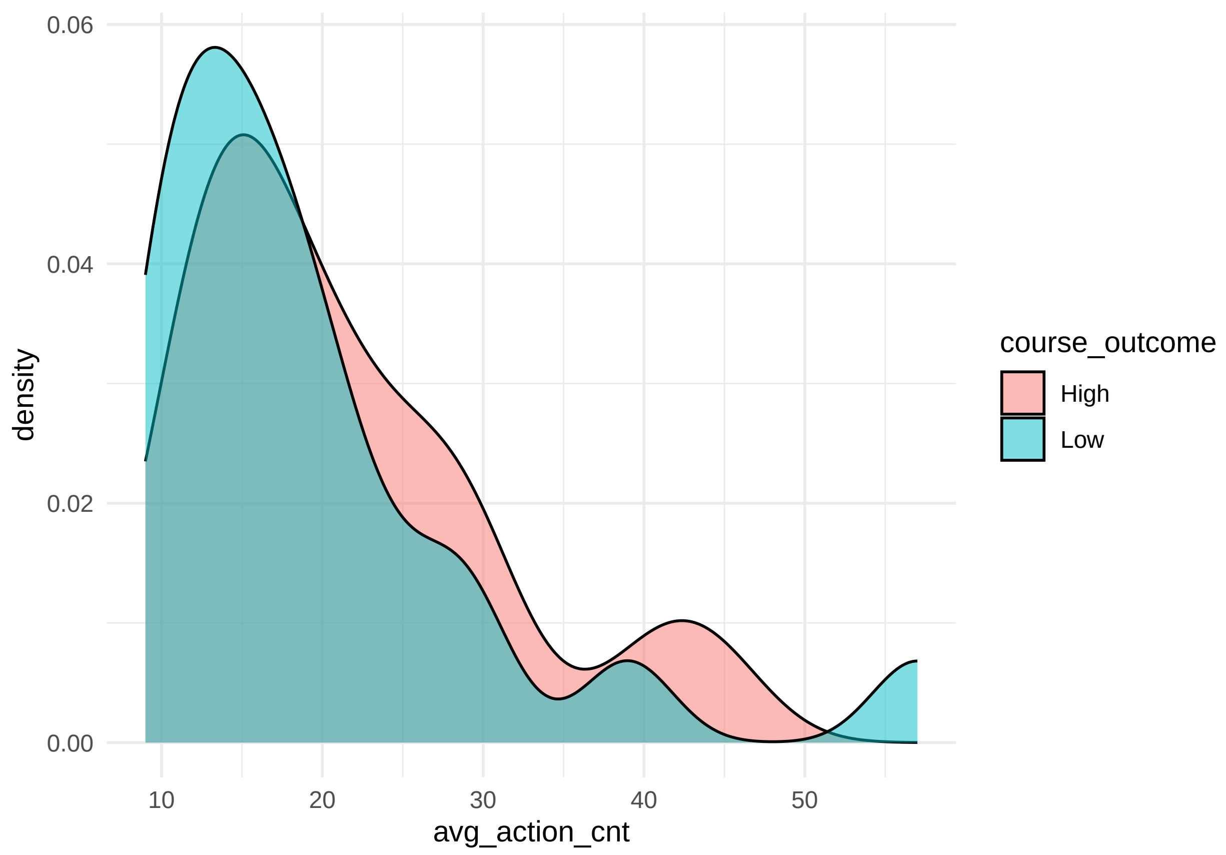
```
feature_names <- colnames(feature_set)[2:11]

# Note: the notation .data[[fn]] in the code below allows us to access a column from the 'current' data
# (in this case, final_ds) with the name given as the input variable of the function (fn)
lapply(feature_names,
       function(fn) {
         ggplot(feature_set, aes(x = .data[[fn]], fill=course_outcome)) +
           geom_density(alpha=0.5) +
           theme_minimal()
       })
```

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



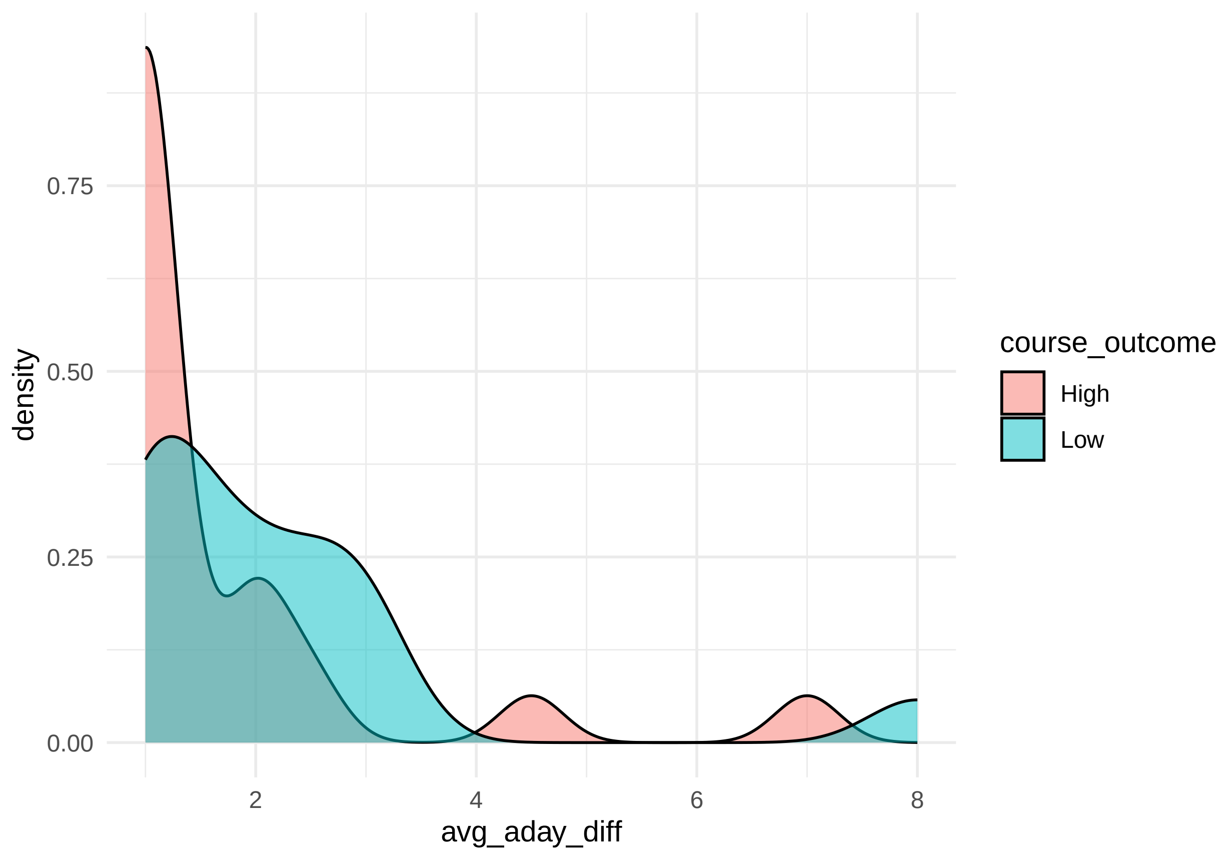
0.00
0.03
0.06
0.09
density
5
10
15
20
adays_cnt
course_outcome
High
Low

```
##
## [[2]]
```

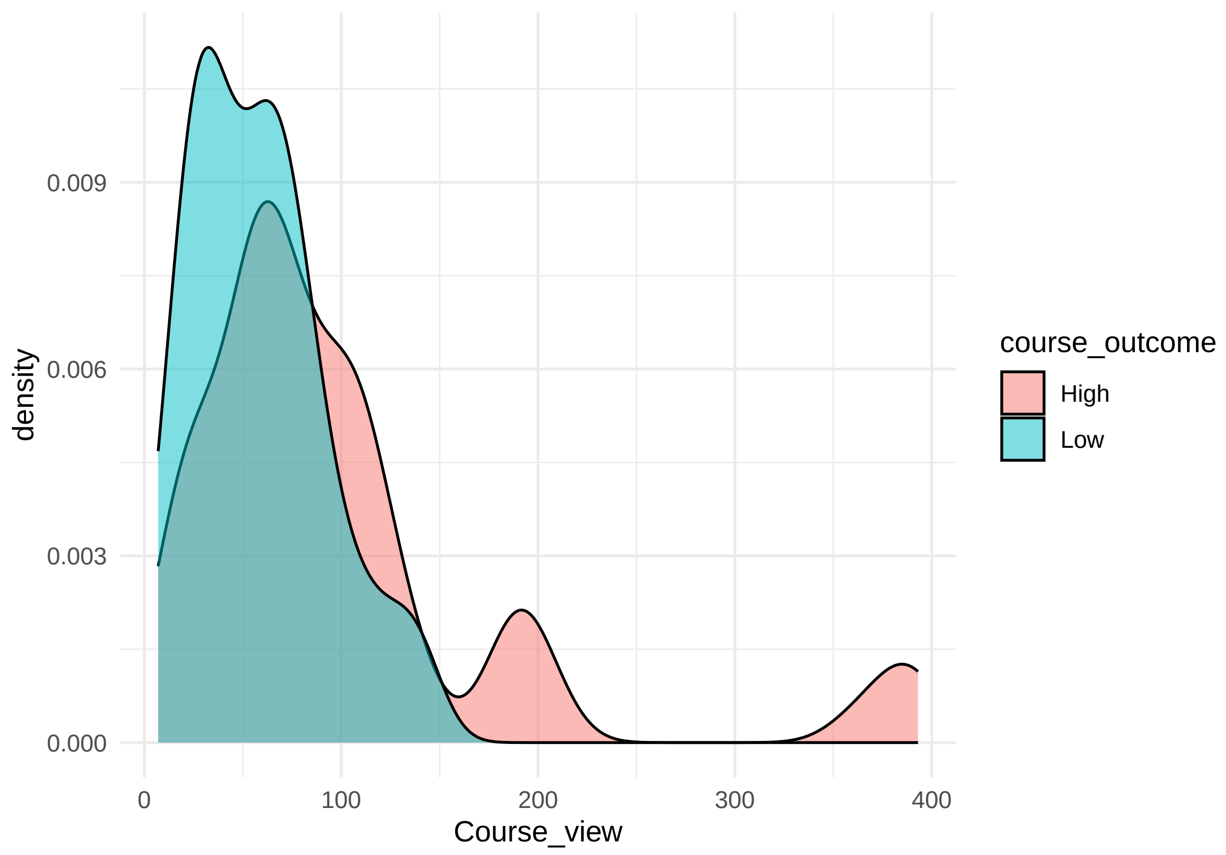


0.06
0.04
0.02
0.00
density
10
20
30
40
50
avg_action_cnt
course_outcome
High
Low

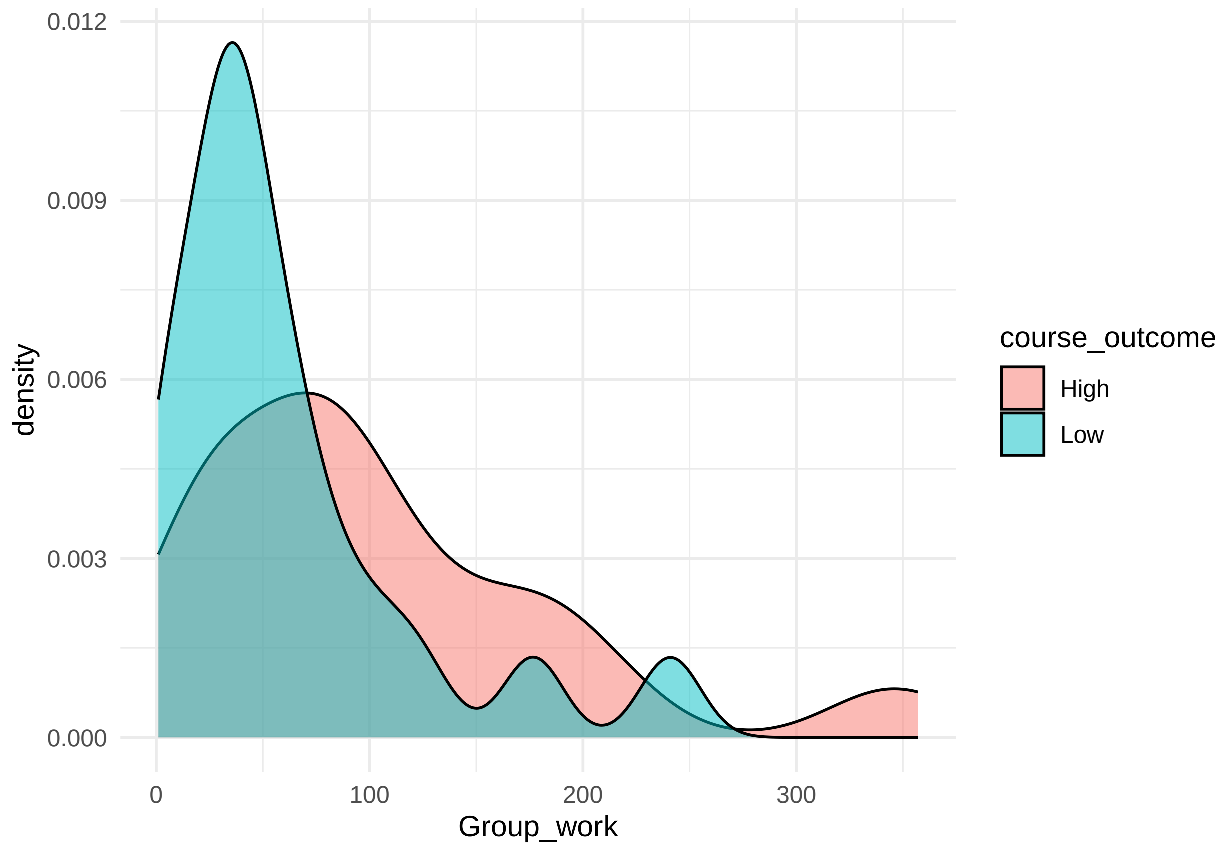
```
##
## [[3]]
```


density
0.75
0.50
0.25
0.00
2
4
6
8
avg_aday_diff
course_outcome
High
Low

```
##
## [[4]]
```

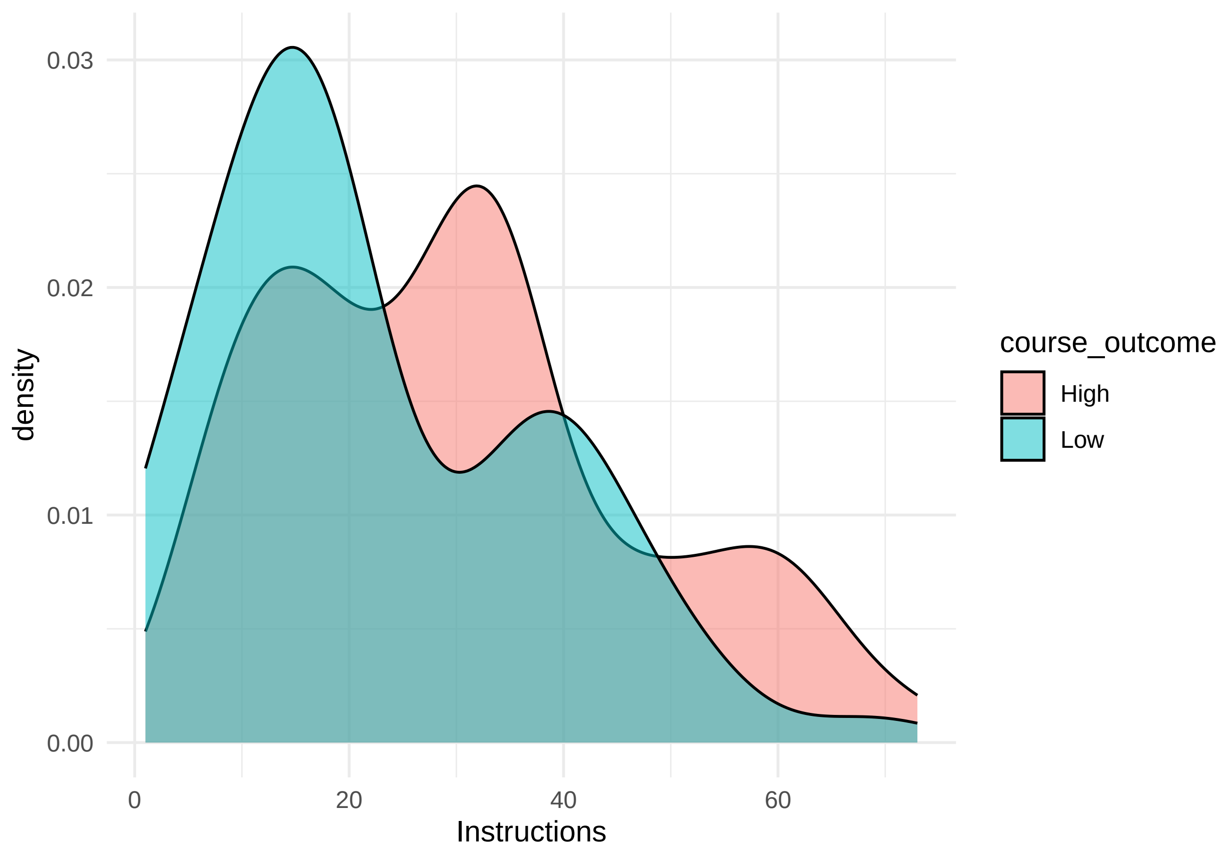

density
0.009
0.006
0.003
0.000
0
100
200
300
400
Course_view
course_outcome
High
Low

```
##
## [[5]]
```

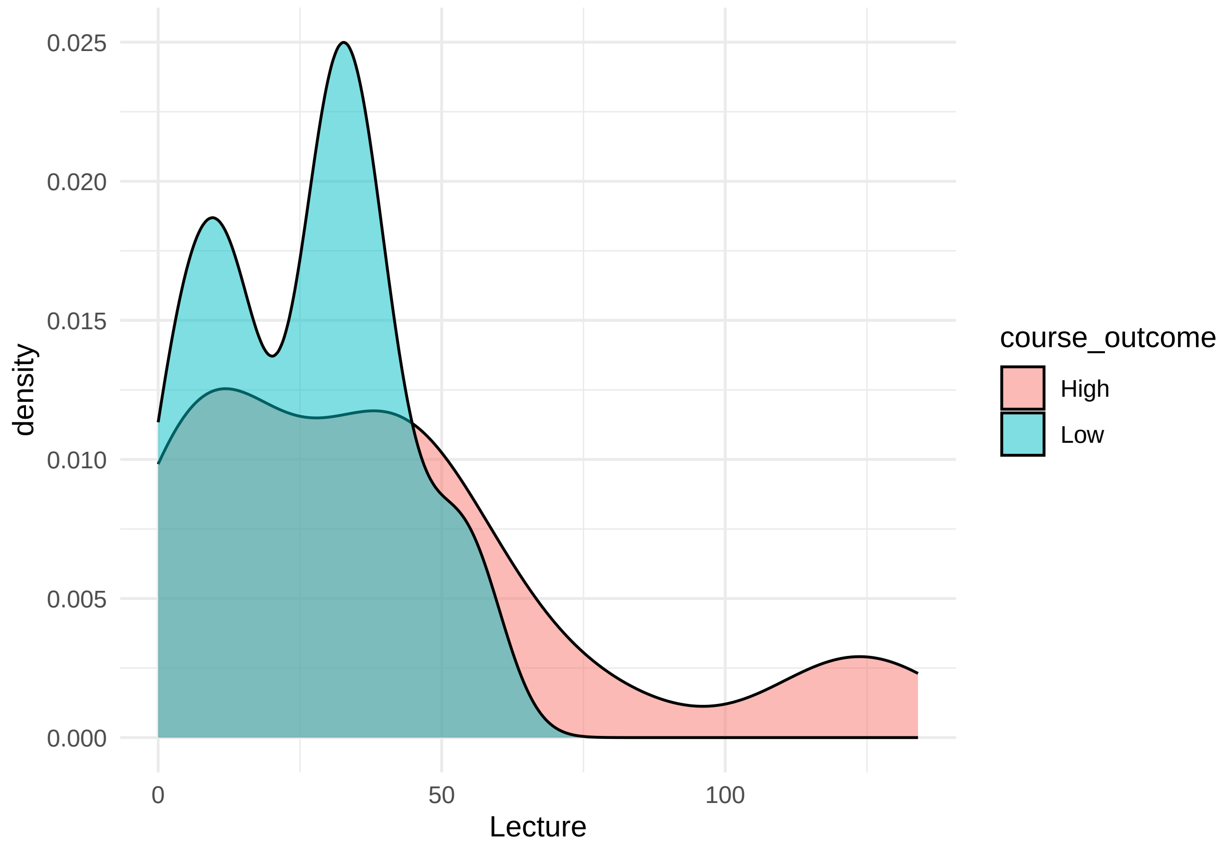


density
0.012
0.009
0.006
0.003
0.000
0
100
200
300
Group_work
course_outcome
High
Low

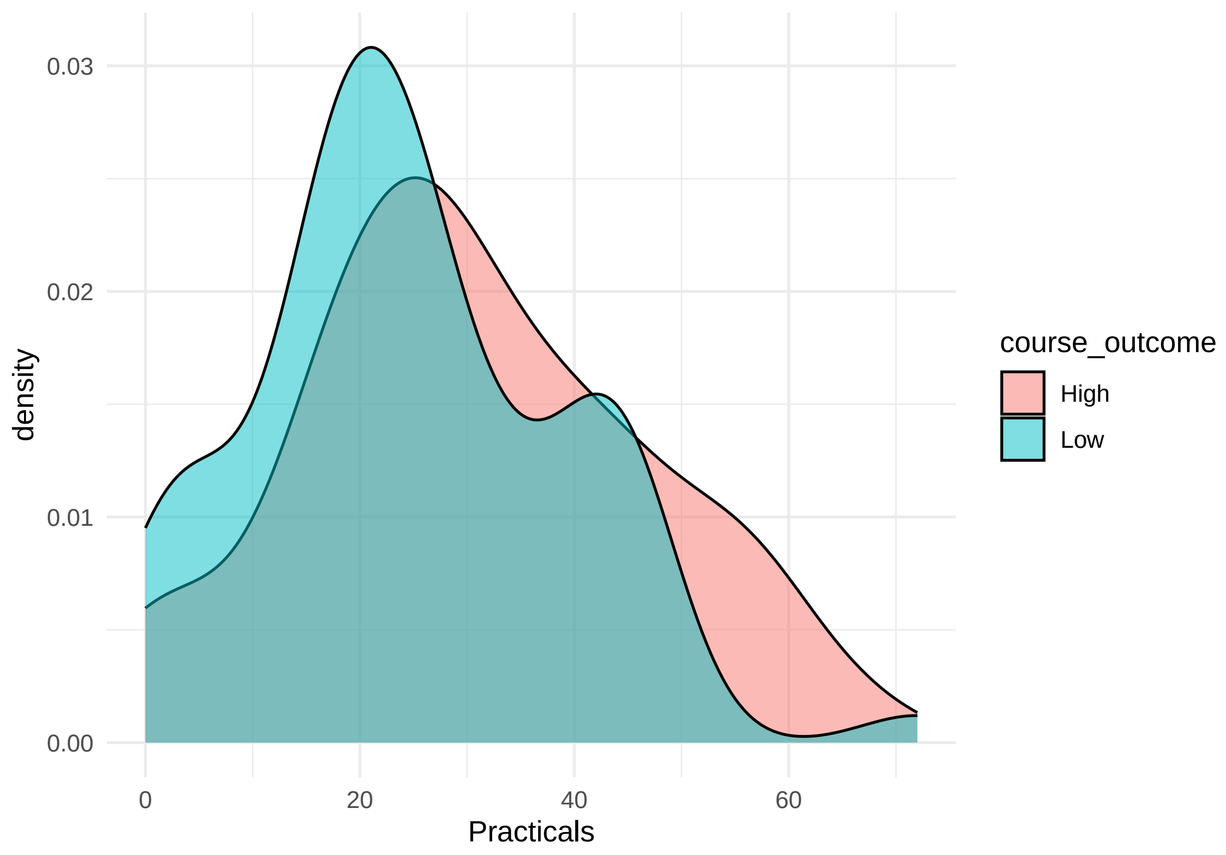
```
##
## [[6]]
```


0.03
0.02
0.01
0.00
density
0
20
40
60
Instructions
course_outcome
High
Low

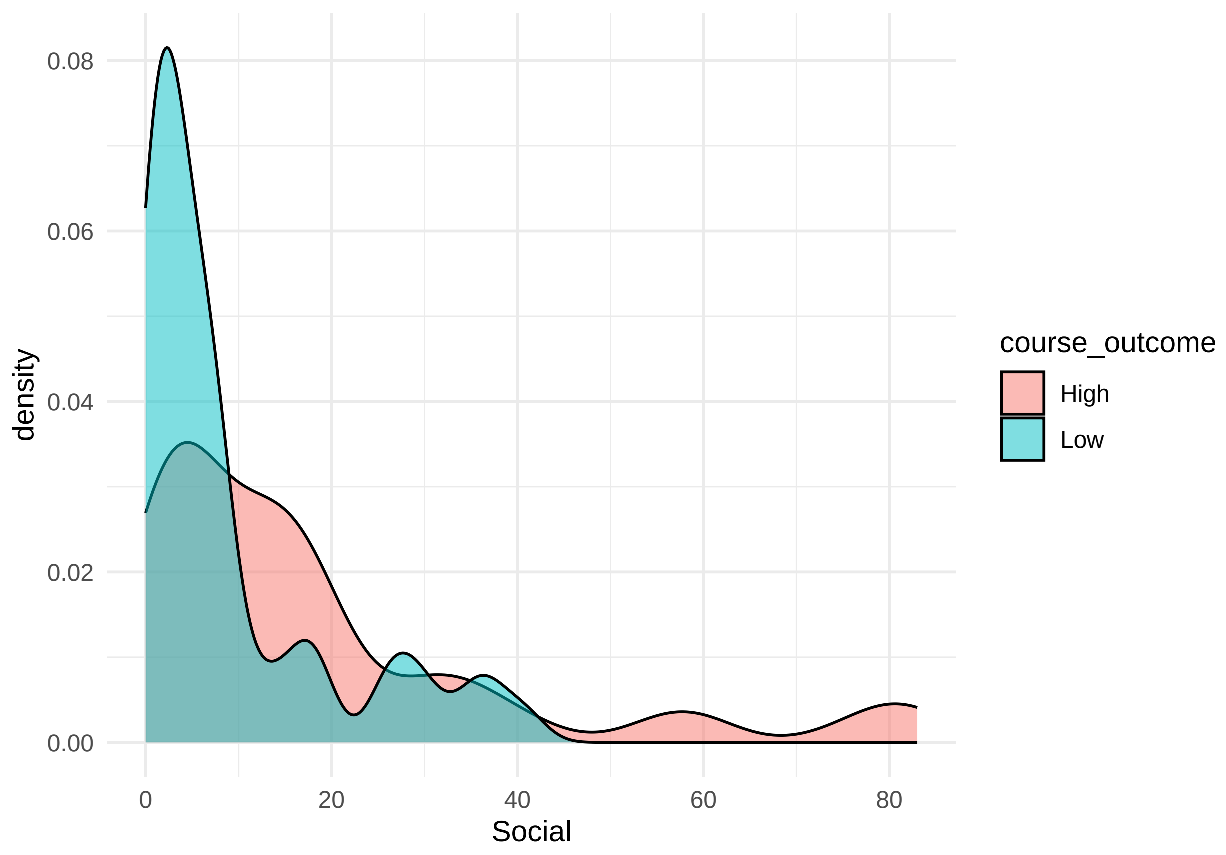
```
##
## [[7]]
```


0.025
0.020
0.015
0.010
0.005
0.000
density
0
50
100
Lecture
course_outcome
High
Low

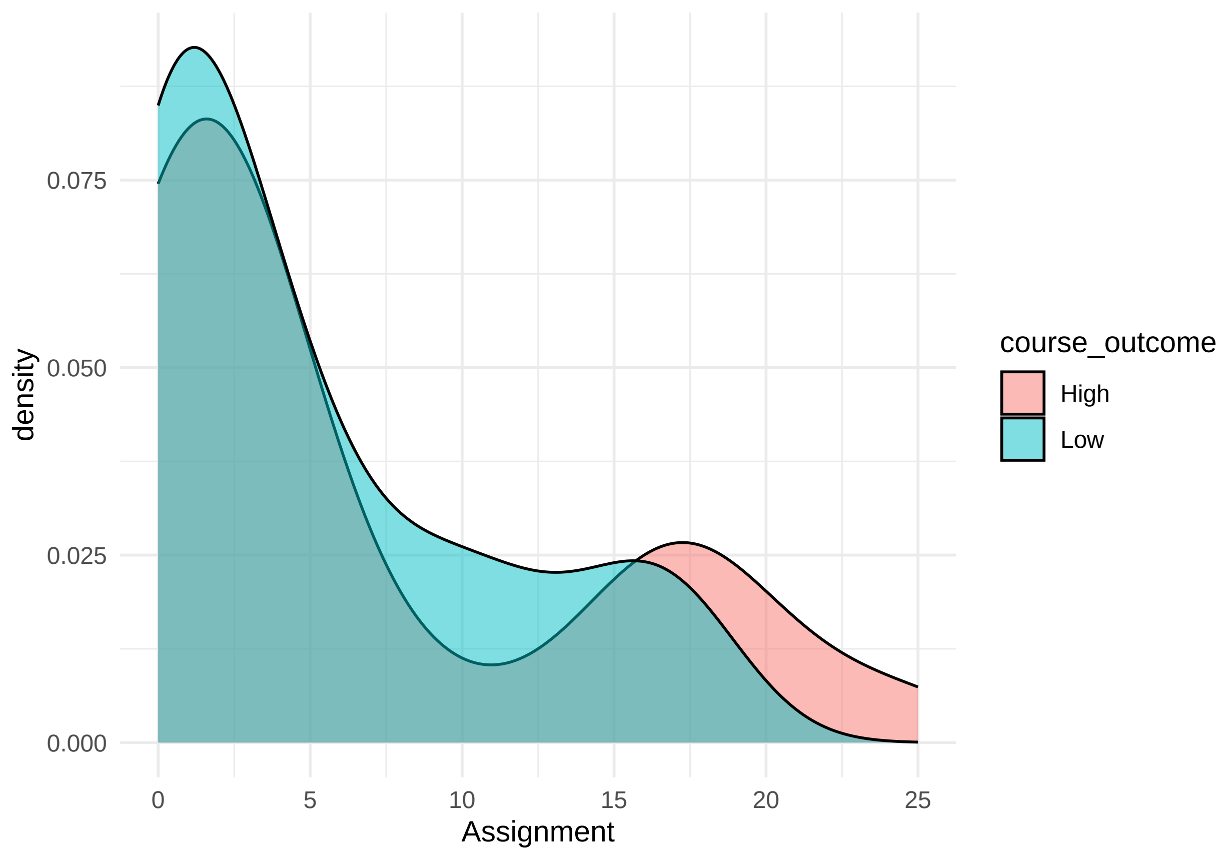
```
##
## [[8]]
```


0.03
0.02
0.01
0.00
density
0
20
40
60
Practicals
course_outcome
High
Low

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


0.08
0.06
0.04
0.02
0.00
density
0
20
40
60
80
Social
course_outcome
High
Low

```
##
## [[10]]
```



The plots suggest that all features - except for `avg_action_cnt`, `Assignment` and maybe `Lecture` - are potentially relevant for predicting the course outcome.

```
feature_set |>
  select(-c(Assignment, avg_action_cnt)) -> final_ds
```

We have now completed the feature engineering and selection work. Since we have a small number of observations and all features are numerical, we can use kNN. As this is something we have done before, the rest of the code will not be commented in detail.

Predictive modeling

Load additional R packages required for model building and evaluation

```
library(caret)
library(class)
```

Recall that kNN requires that all features have roughly equal value ranges. So, we will first check that and do rescalling if value ranges notably differ.

```
summary(final_ds)
```

```
##      user             adays_cnt     avg_aday_diff    Course_view
##  Length:128         Min.   : 3.00   Min.   :1.000   Min.   :  7.00
##  Class :character   1st Qu.: 6.00   1st Qu.:1.000   1st Qu.: 35.75
##  Mode  :character   Median :10.00   Median :1.250   Median : 65.00
##                     Mean   :10.55   Mean   :1.945   Mean   : 79.64
##                     3rd Qu.:14.00   3rd Qu.:2.500   3rd Qu.: 97.25
##                     Max.   :20.00   Max.   :8.000   Max.   :393.00
##    Group_work      Instructions      Lecture          Practicals
##  Min.   :  1.00   Min.   : 1.00   Min.   :  0.00   Min.   : 0.0
##  1st Qu.: 30.75   1st Qu.:14.00   1st Qu.: 11.00   1st Qu.:17.0
##  Median : 58.50   Median :23.00   Median : 30.50   Median :25.0
##  Mean   : 82.87   Mean   :26.48   Mean   : 32.99   Mean   :27.6
##  3rd Qu.:110.50   3rd Qu.:35.25   3rd Qu.: 42.50   3rd Qu.:39.5
##  Max.   :357.00   Max.   :73.00   Max.   :134.00   Max.   :72.0
##      Social      course_outcome
##  Min.   : 0.00   High:65
##  1st Qu.: 2.75   Low :63
##  Median : 7.00
##  Mean   :13.73
##  3rd Qu.:17.25
##  Max.   :83.00
```

Clearly, rescalling is required. We need to determine how to do it…

```
apply(final_ds[,c(2:9)], 2, function(x) length(boxplot.stats(x)$out))
```

```
##     adays_cnt avg_aday_diff   Course_view    Group_work  Instructions
##             0             7             7             7             2
##       Lecture    Practicals        Social
##             7             0             9
```

Considering the presence of outliers, we will do standardization

```
apply(final_ds[,c(2:9)], 2,
      function(x) scale(x, center = median(x), scale = IQR(x))) |> as.data.frame() -> final_rescaled
```

```
summary(final_rescaled)
```

```
##    adays_cnt         avg_aday_diff      Course_view        Group_work
##  Min.   :-0.87500   Min.   :-0.1667   Min.   :-0.9431   Min.   :-0.7210
##  1st Qu.:-0.50000   1st Qu.:-0.1667   1st Qu.:-0.4756   1st Qu.:-0.3480
##  Median : 0.00000   Median : 0.0000   Median : 0.0000   Median : 0.0000
##  Mean   : 0.06836   Mean   : 0.4635   Mean   : 0.2381   Mean   : 0.3055
##  3rd Qu.: 0.50000   3rd Qu.: 0.8333   3rd Qu.: 0.5244   3rd Qu.: 0.6520
##  Max.   : 1.25000   Max.   : 4.5000   Max.   : 5.3333   Max.   : 3.7429
##   Instructions        Lecture            Practicals          Social
##  Min.   :-1.0353   Min.   :-0.96825   Min.   :-1.1111   Min.   :-0.4828
##  1st Qu.:-0.4235   1st Qu.:-0.61905   1st Qu.:-0.3556   1st Qu.:-0.2931
##  Median : 0.0000   Median : 0.00000   Median : 0.0000   Median : 0.0000
##  Mean   : 0.1640   Mean   : 0.07912   Mean   : 0.1156   Mean   : 0.4639
##  3rd Qu.: 0.5765   3rd Qu.: 0.38095   3rd Qu.: 0.6444   3rd Qu.: 0.7069
##  Max.   : 2.3529   Max.   : 3.28571   Max.   : 2.0889   Max.   : 5.2414
```

Add the outcome variable to complete the data set

```r
final_rescaled$course_outcome <- final_ds$course_outcome
```

Split the data into training and test sets

```r
set.seed(2024)
for_training = createDataPartition(final_rescaled$course_outcome, p = 0.8, list = FALSE)
train_data = final_rescaled[for_training,]
test_data = final_rescaled[-for_training,]
```

Use cross-validation to determine the optimal value for K (the number of nearest neighbours to consider)

```r
k_grid <- expand.grid(.k = seq(3, 15, 2))
ctrl <- trainControl(method = "CV", number = 10)

set.seed(2024)
knn_cv <- train(x = train_data[,-9],
                y = train_data$course_outcome,
                method = "knn",
                tuneGrid = k_grid,
                trControl = ctrl)

knn_cv
```

```
## k-Nearest Neighbors
##
## 103 samples
##   8 predictor
##   2 classes: 'High', 'Low'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 91, 93, 93, 93, 93, 93, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    3  0.6721212  0.3468817
##    5  0.6146970  0.2304372
##    7  0.5863636  0.1694915
##    9  0.6245455  0.2467797
##   11  0.6145455  0.2267797
##   13  0.5945455  0.1867797
##   15  0.6336364  0.2666667
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 3.
```

Make predictions on the test set using the optimal value for K

```r
knn_pred = knn(train = train_data[,-9],
               test = test_data[,-9],
               cl = train_data$course_outcome,
               k = knn_cv$bestTune$k)
```

Evaluate the predictions using the standard set of evaluation measures

```r
source("util.R")
```

```r
cm <- table(true = test_data$course_outcome,
            predicted = knn_pred)

cm
```

```
##       predicted
## true   High Low
##   High   11   2
##   Low     3   9
```

```r
compute_eval_measures_v1(cm)
```

```
##  accuracy precision    recall        F1
##    0.8000    0.8182    0.7500    0.7826
```

The results suggest that already early in this course, it is possible to fairly well predict students at risk of low performance (positive class in this case). For example, out of all those who eventually showed low course performance, we have correctly identified 75% of them. Likewise, when predicting someone as having low course performance, we were correct in 82% of cases.