Practicum 1

Josiah Parry

2020-01-02

Problem 1

(0 pts) Download the data set Glass Identification Database along with its explanation. Note that the data file does not contain header names; you may wish to add those. The description of each column can be found in the data set explanation. This assignment must be completed within an R Markdown Notebook.

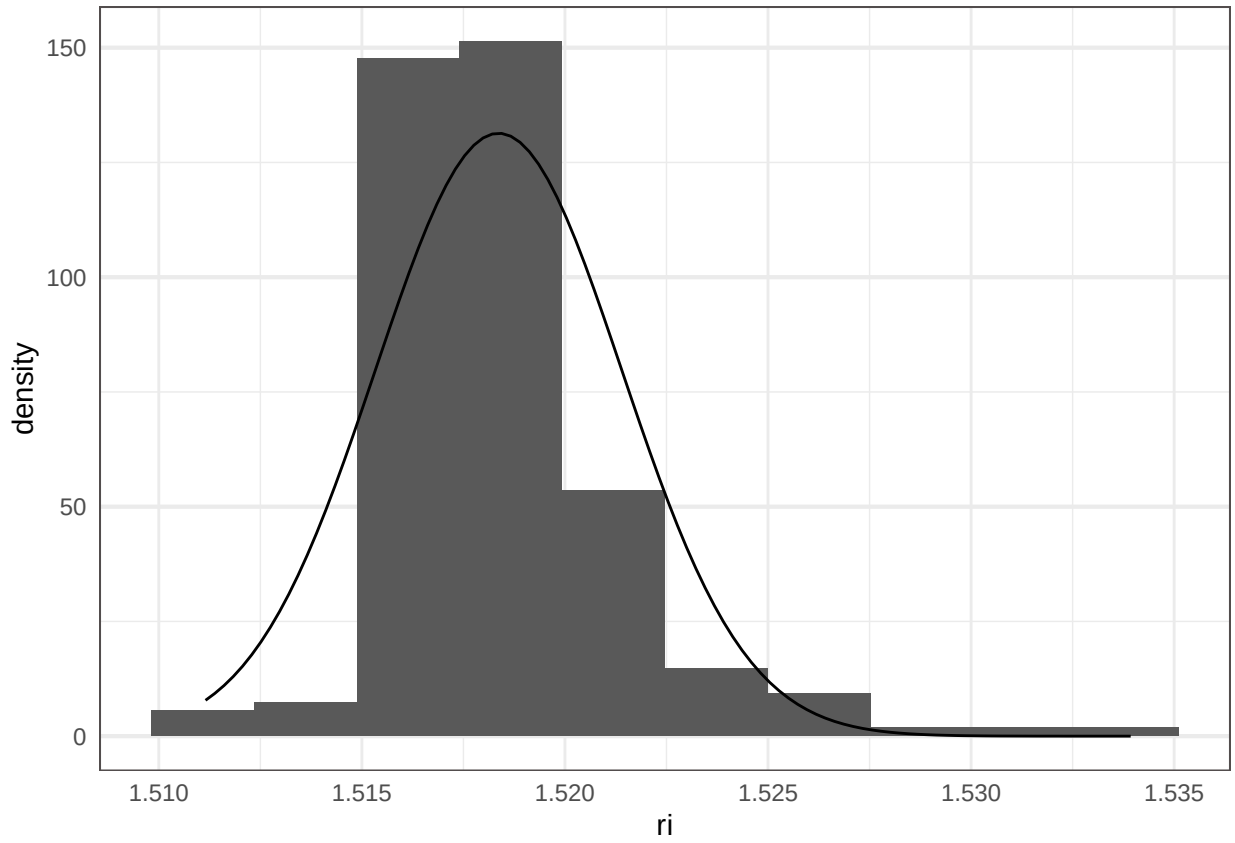
(0 pts) Explore the data set as you see fit and that allows you to get a sense of the data and get comfortable with it.

```
options(scipen = 17)
library(tidyverse)
library(rsample)
library(recipes)

# create vector of column names
colnames <- c("id", "ri", "na", "mg", "al", "si", "k", "ca", "ba", "fe", "type")

# read in the csv
glass <- read_csv("data/glass.data",
                  col_names = colnames)

# skim the data for visual of distributions and missingness
# commented out because the `hist` won't knit to pdf
# skimr::skim_to_list(glass) %>%
#   pluck(1) %>%
#   knitr::kable()
```

(5 pts) Create a histogram of column 2 (refractive index) and overlay a normal curve; visually determine whether the data is normally distributed.

Utilize `ggplot2` to create a histogram with an overlain normal distribution.

```
# plotting hist over fr
ggplot(glass, aes(ri)) +
  geom_histogram(aes(y = ..density..), bins = 10) +
  stat_function(fun = dnorm,
                args = list(mean = mean(glass$ri),
                            sd = sd(glass$ri)))
```



Conduct a shapiro test for normality.

```
shapiro.test(glass$ri)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  glass$ri
## W = 0.86757, p-value = 0.000000000001077
```

The shapiro test returns a p value < 0.001. We can then conclude that the distribution is in fact not normaly distributed.

> (5 pts) Does the k-NN algorithm require normally distributed data or is it a non-parametric method? Comment on your findings.

k-nearest neighbors is a non parametric model that does not require normally distributed data. There is only one hyper-parameter in this model and that is k. We determine k and no parameters are discovered. One should note that distances can be affected by the magnitude of the variables and as such input vectors should be rescaled in some manner.

> (5 pts) Identify any outliers for the columns using a z-score deviation approach, i.e., consider any values that are more than 2 standard deviations from the mean as outliers. Which are your outliers for each column? What would you do? Do not remove them the outliers.

```r
outs <- glass %>%
  # standardize all vars
  mutate_at(vars(everything(), -id), scale) %>%
  # reshape
  pivot_longer(cols = -id, names_to = "var", values_to = "z") %>%
  filter(abs(z) > 2)

knitr::kable(count(outs, var, sort = TRUE))
```

var	n
type	29
ba	15
al	12
ca	12
fe	12
ri	12
si	12
na	9
k	3

Following a z-score approach, there are a total of 116 outliers. Outliers should not be removed from the data. We have no valid explanation of *why* they should be removed. They may very well be properly observed data and are in fact representative of the underlying distribution. Outliers should only be observed if a strong case for their removal can be made such as measurement error.

> (10 pts) After removing the ID column (column 1), normalize the numeric columns, except the last one, using z-score standardization. The last column is the glass type and so it is excluded.

The is done in the recipe prep stage below. =

> (10 pts) The data set is sorted, so creating a validation data set requires random selection of elements. Create a stratified sample where you randomly select 20% of each of the cases for each glass type to be part of the validation data set. The remaining cases will form the training data set.

knn is not a trained model. This question is misleading. There is no "validation" of a knn model. A knn model holds all data in memory and is the reason why it is not recommended at large scale. One must recalculate distances with each new observation.

In the training recipe we keep both the `type` and `id` columns for later use and validation.

```r
# stratified sample
init_split <- initial_split(
  glass,
  prop = .8,
  strata = type
)

# extract the training set
train_df <- training(init_split)
```

```r
# prepping recipe
train_rec <- recipe(type ~., data = train_df) %>%
  # center and scale - i.e. "z-score normalization"
  step_center(all_numeric(), -type, -id) %>%
  step_scale(all_numeric(), -type, -id) %>%
  prep()
```

(20 pts) Implement the k-NN algorithm in R (do not use an implementation of k-NN from a package) and use your algorithm with a k=6 to predict the glass type for the following two cases:

First, I create a function that calculates the euclidian distance for all observations.

```r
tidy_dist <- function(df, id_col) {

  # calculate the euclidian distances
  mdist <- select(df, -{{ id_col }}) %>%
    dist(diag = TRUE, upper = FALSE) %>%
    as.matrix(labels = TRUE)

  # reshape the distance matrix into a dataframe
  ret <- as.data.frame(mdist) %>%
    setNames(pull(df, id)) %>%
    mutate(id = pull(df, id)) %>%
    gather(item2, distance, -id) %>%
    as_tibble()

  # remove upper triangle
  # this removes duplicate values
  ret[!upper.tri(mdist),] %>%
    # remove diag
    # removes distance from self
    filter(distance != 0)

}
```

Next, we need to apply the normalization steps to the data.

```r
baked_train <- bake(train_rec, train_df)
```

Then I calculate distances for all observations using this new function.

```r
glass_dist <- tidy_dist(baked_train, id)
```

Then, to identify which type each observation actually belongs to, I join it back to the original data set using the `id` column. I select only the `id` and `type` columns for simplicity and data reduction. Following, I identify the 6 observations with the smalled euclidian distance then assign a predicted class based on majority vote.

```r
# define mode function because base R
# still doesn't have mode??!?! bah.
mode <- function(x) {
  ux <- unique(x)
  ux[which.max(tabulate(match(x, ux)))]
}
```

```
# joining distance matrix back onto original data
preds <- left_join(glass_dist,
          select(glass, id, type)) %>%
  group_by(item2) %>%
  # filtering to only the 6 closest neighbors
  top_n(-6, distance) %>%
  # identify most observed class
  summarise(class = mode(type)) %>%
  mutate(item2 = as.integer(item2)) %>%
  left_join(select(glass, id, type),
            by = c("item2" = "id"))
```

```
## Joining, by = "id"
```

I visualize the confusion matrix with some help from `janitor`.

```
janitor::tabyl(preds, class, type)
```

```
##  class  1  2  3 5 6  7
##      1 42  0  0 0 0  0
##      2 10 56  0 0 0  0
##      3  0  7 11 0 0  0
##      5  0  2  2 3 0  0
##      6  0  0  1 4 1  0
##      7  0  0  0 2 5 25
```

I calculate the accuracy below.

```
mean(preds$class == preds$type)
```

```
## [1] 0.8070175
```

> Use the whole normalized data set for this; not just the training data set. Note that you need to normalize the values of the new cases the same way as you normalized the original data.

In order to make predictions with the below observations I create a function to do so.

```
classify_type <- function(new_data) {

  # create new data frame with training data and new data
  bind_rows(
    # pre-process new data
    bake(train_rec, new_data),
    # pre-process glass
    bake(train_rec, glass)
  ) %>%
    # calc dist
    tidy_dist(id) %>%
    # join glass back on for type
    left_join(select(glass, id, type), by = "id") %>%
```

```
    # group for counts
    group_by(item2) %>%
    # identify 6 neighbors
    top_n(-6, distance)  %>%
    # identify the pred class
    summarise(class = mode(type)) %>%
    # identify only the new dat
    filter(item2 == 999) %>%
    # pull out the underlying vector
    pull(class)

}
```

RI = 1.51621 | 12.53 | 3.48 | 1.39 | 73.39 | 0.60 | 8.55 | 0.00 | Fe = 0.08 RI = 1.5893 | 12.71 | 1.85 | 1.82 | 72.62 | 0.52 | 10.51 | 0.00 | Fe = 0.05

I create a new data frame with the above data. I assign the `id` value to `999` so I can identify these later.

```
new_dat <- tribble(
  ~"ri", ~"na", ~"mg", ~"al", ~"si", ~"k", ~"ca", ~"ba", ~"fe",
  1.51621 , 12.53 , 3.48 , 1.39 , 73.39 , 0.60 , 8.55 , 0.00 , 0.08,
  1.5893 , 12.71 , 1.85 , 1.82 , 72.62 , 0.52 , 10.51 , 0.00 , 0.05
) %>%
  mutate(id = 999)
```

I create the predictions using `purrr`. We must iterate because if we add other observations it creates an inacurate distance measure for the single observation.

```
map_dbl(1:2, ~{
  classify_type(
    slice(new_dat, .x)
    )
  }
)
```

```
## [1] 1 2
```

(10 pts) Apply the knn function from the class package with k=6 and redo the cases from Question (7). Compare your answers.

Below I use the `class::knn()` classifier to generate predictions.

```
library(class)

baked_test <- testing(init_split) %>%
  bake(train_rec, .)

class_preds <- knn(
  train = select(baked_train, -id, -type),
  test = select(baked_test, -type, -id),
  cl = pull(baked_train, type),
  k = 6)
```

I define my own testing set and perform my own predictions below. Note that the preprocessing occurs within the `classify_type()` function that was defined previously.

```
testing_set <- testing(init_split) %>%
  mutate(id = 999)

my_preds <- map_dbl(1:nrow(testing_set), ~{
  classify_type(
    slice(testing_set, .x))
  }
)
```

I calculate the accuracy of each classifier.

```
mean(class_preds == baked_test$type)
```

```
## [1] 0.452381
```

```
mean(my_preds == baked_test$type)
```

```
## [1] 0.8571429
```

(10 pts) Using your own implementation as well as the class package implementation of kNN, create a plot of k (x-axis) from 2 to 10 versus error rate (percentage of incorrect classifications) for both algorithms using ggplot.

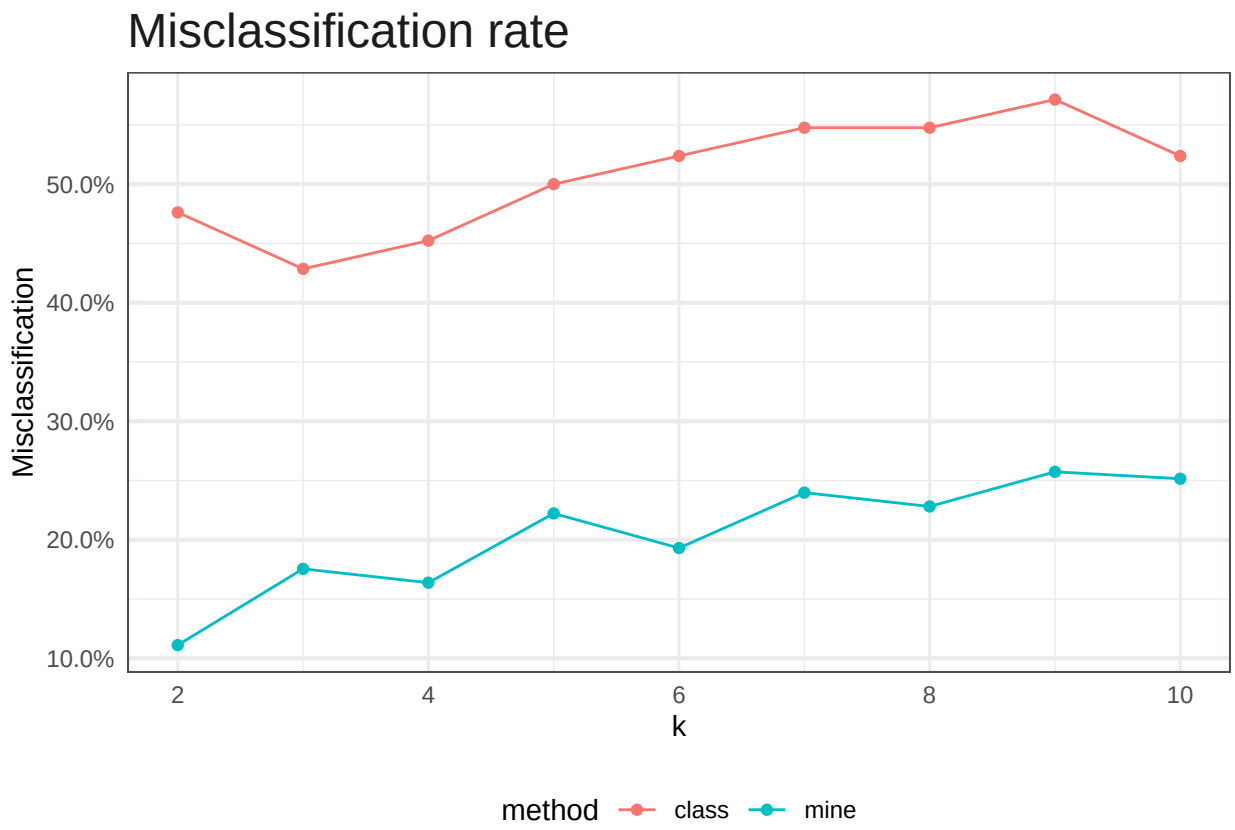
```
# create function for making predictions and calc acc
glass_knn <- function(k) {
  preds <- left_join(glass_dist,
                     select(glass, id, type)) %>%
    group_by(item2) %>%
    top_n(-k, distance) %>%
    summarise(class = mode(type)) %>%
    mutate(item2 = as.integer(item2)) %>%
    left_join(select(glass, id, type),
              by = c("item2" = "id"))
  # calculate acc and "error"
  # should really be specificity and sensitivity
  preds %>%
    summarise(accuracy = mean(class == type),
              misclass = 1 - accuracy)

}
```

```
# create a tibble of predictions for k [2, 10] with class::knn
k_preds <- map_dfr(2:10, ~{
  class_preds <- knn(
    train = select(baked_train, -id, -type),
    test = select(baked_test, -type, -id),
    cl = pull(baked_train, type),
    k = .x)
```

```
  tibble(
    k = .x,
    accuracy = mean(class_preds == baked_test$type),
    misclass = 1 - accuracy,
    method = "class"
  )

}) %>%
  # create tibble of predictions for k [2, 10] using my own function
  bind_rows(
    map_dfr(2:10, glass_knn, .id = "k") %>%
      mutate(k = as.integer(k) + 1,
             method = "mine")
  )
```

```
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
## Joining, by = "id"
```

```
# visualize "error"
k_preds %>%
  ggplot(aes(k, misclass, color = method)) +
  geom_line() +
  geom_point() +
  labs(title = "Misclassification rate",
       y = "Misclassification") +
  scale_y_continuous(labels = scales::percent) +
  theme(legend.position = "bottom")
```



(10 pts) Produce a cross-table confusion matrix showing the accuracy of the classification using knn from the class package with k = 5.

```
class_preds <- knn(
  train = select(baked_train, -id, -type),
  test = select(baked_test, -type, -id),
  cl = pull(baked_train, type),
  k = 5)

table(class_preds, baked_test$type)
```

```
##
## class_preds  1  2  3  5  6  7
##           1 11  5  2  0  0  0
##           2  7  4  1  2  0  1
##           3  0  0  0  0  0  0
##           5  0  1  0  1  0  0
##           6  0  1  0  0  2  0
##           7  0  0  0  1  1  2
```

(10 pts) Download this (modified) version of the Glass data set containing missing values in column 4. Identify the missing values. Impute the missing values using your version of kNN from Problem 2 below using the other columns are predictor features.

I am unsure specifically what is being asked. I am going to work under the assumption that it is desired to impute the `mp` values into the newly downloaded data using our self-created KNN algorithm where the "training" data is the full glass data set and the "test" set is the missing values.

Since the type of imputation was not specified I will impute the average `mg` value.

```
new_glass <- read_csv("data/glass_missing.csv",
                      col_names = colnames) %>%
  mutate(id = id + 900)
```

```
## Parsed with column specification:
## cols(
##   id = col_double(),
##   ri = col_double(),
##   na = col_double(),
##   mg = col_double(),
##   al = col_double(),
##   si = col_double(),
##   k = col_double(),
##   ca = col_double(),
##   ba = col_double(),
##   fe = col_double(),
##   type = col_double()
## )
```

```
# skim data to find missingness
skimr::skim(new_glass)
```

```
## Skim summary statistics
##  n obs: 214
##  n variables: 11
##
## -- Variable type:numeric -------------------------------------------------------------------
##  variable missing complete   n     mean     sd     p0    p25     p50
##        al       0      214 214    1.44   0.5     0.29   1.19    1.36
##        ba       0      214 214    0.18   0.5     0      0       0
##        ca       0      214 214    8.96   1.42    5.43   8.24    8.6
##        fe       0      214 214    0.057  0.097   0      0       0
##        id       0      214 214 1007.5   61.92  901    954.25 1007.5
##         k       0      214 214    0.5    0.65    0      0.12    0.56
##        mg       9      205 214    2.73   1.41    0      2.24    3.48
##        na       0      214 214   13.41   0.82   10.73  12.91   13.3
##        ri       0      214 214    1.52   0.003   1.51   1.52    1.52
##        si       0      214 214   72.65   0.77   69.81  72.28   72.79
##      type       0      214 214    2.78   2.1     1      1       2
##      p75    p100     hist
##     1.63    3.5
##     0       3.15
##     9.17   16.19
##     0.1     0.51
##  1060.75 1114
##     0.61    6.21
##     3.61    4.49
```

```
##    13.83   17.38
##     1.52    1.53
##    73.09   75.41
##     3       7
```

```
# create df with only missing data
mg_missing <- filter(new_glass, is.na(mg))
```

Below I create a function to identify k neighbors.

```
# create general function to find the neighbors
find_neighbors <- function(new_data, k) {
  new_data_ids <- new_data$id
  # create new data frame with training data and new data
  bind_rows(
    # pre-process new data
    bake(train_rec, new_data),
    # pre-process glass
    bake(train_rec, glass)
  ) %>%
    # calc dist
    tidy_dist(id) %>%
    # join glass back on for type
    left_join(select(glass, id, type), by = "id") %>%
    filter(item2 %in% new_data_ids) %>%
    # group for counts
    group_by(item2) %>%
    # identify 6 neighbors
    top_n(-k, distance)  %>%
    arrange(desc(item2)) %>%
    select(neighbor = id, input_id = item2) %>%
    select(2, 1)
}
```

Below I iterate over the missing observations and find the 6 nearest neighbors.

```
# iterate over missing rows
mg_neighbors <- map_dfr(1:nrow(mg_missing), ~{
  find_neighbors(
    slice(mg_missing, .x),
    6)
}
)
```

I then find the average `mg` value for the 6 neighbors.

```
# take 6 closest neighbors and calculate mean mg values
imputed_mg <- left_join(mg_neighbors, glass, by = c("neighbor" = "id")) %>%
  group_by(input_id) %>%
  summarise(mg = mean(mg))
```

The last step is to put the imputed data back into the dataframe.

```
# bind rows and all is well.
all_imputed <- mg_missing %>%
  mutate(mg = pull(arrange(imputed_mg, as.integer(input_id)), mg)) %>%
  bind_rows(
    filter(new_glass, !id %in% pull(mg_missing, id))
    )
```

Problem 2

```
library(data.table)
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
```

```
## The following object is masked from 'package:purrr':
##
##     transpose
```

```
library(tidyverse)
library(rsample)
library(recipes)
```

Problem 2

1. (0 pts) Investigate this data set of home prices in King County (USA).

2. (5 pts) Save the price column in a separate vector/dataframe called target_data. Move all of the columns except the ID, date, price, yr_renovated, zipcode, lat, long, sqft_living15, and sqft_lot15 columns into a new data frame called train_data.

```
housing <- read_csv("data/kc_house_data.csv") %>%
  select(-c(id, date, yr_renovated, zipcode, lat, long, sqft_living15, sqft_lot15))
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   id = col_character(),
##   date = col_datetime(format = "")
## )
```

```
## See spec(...) for full column specifications.
```

3. (5 pts) Normalize all of the columns (except the boolean columns waterfront and view) using min-max normalization.

As this is extremely computationally intensive to calculate the distances, I will do this on a sample.

```
house_rec <- recipe(price ~ ., data = housing,
                    retain = TRUE) %>%
  step_range(contains("sqft"),
             min = 0, max = 1) %>%
  prep()

# extract the pre-processed data
house_pp <- juice(house_rec) %>%
  mutate(id = row_number())

# split the data into 50%
house_split <- initial_split(house_pp, .5)
```

The below code chunk is similar to the `tidy_dist()` function above but it utilizes `data.table` rather than `tidyr` for data reshaping. This is done for performance puproses. Manipulating matrixes is a very computationally intesive exercise. In order for this to be performative, C++ code would need to be written. As noted previously, KNN does not scale well.

```
# define distance function using data.table
dt_dist <- function(df, id_col) {

  # calculate the euclidian distance between all observations
  # cast as a matrix
  mdist <- select(df, - {{ id_col }}) %>%
    dist(diag = TRUE, upper = FALSE) %>%
    as.matrix(labels = TRUE)

  # convert the matrix to a data.table
  # reshape into tidy format with `melt()`
  # cast back into a tibble
  ret <- as.data.table(mdist) %>%
    setNames(as.character(pull(df, {{ id_col }}))) %>%
    .[, id := pull(df, {{ id_col }})] %>%
    setkey(id) %>%
    melt(id.vars = "id",
         variable.name = "item2",
         value.name = "distance") %>%
    as_tibble()

  # remove the lower triangle
  ret[!upper.tri(mdist),] %>%
    # remove diag
    filter(distance != 0)
}
```

The below code chunk creates a new data frame with the required field.

```
new_df <- tibble(
  bedrooms = 4,
  bathrooms = 3,
  sqft_living = 4852,
  sqft_lot = 10244,
  floors = 3,
  waterfront = 0,
  view = 1,
  condition = 3,
  grade = 11,
  sqft_above = 1960,
  sqft_basement = 820,
  yr_built = 1978
)
```

4. (15 pts) Build a function called knn.reg that implements a regression version of kNN that averages the prices of the k nearest neighbors using a weighted average where the weight is 3 for the closest neighbor, 2 for the second closest and 1 for the remaining neighbors (recall that a weighted average requires that you divide the sum product of the weight and values by the sum of the weights).

The below code chunk creates a function for calculating the k nearest neighbors and returns the weighted average of the specified `target` column.

Note that using a period in a function name denotes the use of an S3 class and as such can create issues in class differentiation. For this reason I will not use a period in the function name. Please see the Syntax section of the R Style Guide.

Addionally, I've reordered and renamed the function arguments to be more informative and instinctually positioned. The function takes `target` as the unquoted column name from `train` and utilized `{{ }}` for tidy evaluation.

```
# The data needs to be pre-processed
# I'm providing the recipe object inside of the function
# this makes it rely on external objects.
# This is not good practice. You _must_ run the above code first
knn_reg <- function(new_data, train, target, k) {

  # apply the pre-processing steps to the new data and "training" data
  # create an ID column as needed for the distance formula
  full_df <- bake(house_rec, new_data) %>%
    mutate(id = -999) %>%
    bind_rows(
      bake(house_rec, train) %>%
        mutate(id = row_number())
    )

  # calculate euclidean distance
  house_dist <- dt_dist(full_df, id)

  # find k nearest neighbors
  top_k <- house_dist %>%
    # removing factor class as it slows it down
    mutate(item2 = as.character(item2)) %>%
```

```
    group_by(item2) %>%
    # grab the k nears neighbors based on distance measure
    top_n(-k, distance) %>%
    # calculating the descending rank which will be used as the weight
    mutate(rank = frankv(distance, order = -1))

  top_k %>%
    # join back to original data to get target column values from
    # the "train" data
    left_join(full_df, by = "id") %>%
    # identify only the `new_data`
    filter(item2 == -999) %>%
    # calculate the weighted average and the unweighted average
    summarise(wgt_estimate = sum(({{ target }} * rank)) / sum(rank),
           estimate = mean(price))
}
```

5. (5 pts) Forecast the price of this new home using your regression kNN using k = 4:

Identify the weighted average of `price` based on the 4 nearest neighbors for the `new_df`.

```
pred <- knn_reg(new_df, training(house_split), price, 4)

pred
```

```
## # A tibble: 1 x 3
##   item2 wgt_estimate estimate
##   <chr>        <dbl>    <dbl>
## 1 -999        934000   968750
```

Recreating the environment

If your machine has outdated R packages, try restoring the environment used in this project. To do so, install the `renv` package from CRAN and restore the package environment from the `renv.lock` file.

To do so, I would recommend copying the R project.

Navigate `File > New Project > Version Control > Git` and put in the URL `https://github.com/JosiahParry/da5030`

Once in the project navigate to the console and enter

```
renv::init()
renv::restore()
```