Homework 1

## Question 2.1

Today is Tuesday, May, 15, 2018, and let’s say that I’m a professional sports gambler. (I’m not really, but I’ve always thought it might be cool to be one.) The National Basketball Association (NBA) playoffs are going on right now, and the Cleveland Cavaliers are playing the Boston Celtics in a seven-game series (the Eastern Conference Finals). Cleveland lost Game 1 at Boston two days ago, and the two teams are about to play again tonight in Boston in Game 2. I see that the sports books have made this game a “pick-em” with -110 odds, on both sides. (This means that I must pay $110 to win $100 if I bet either side wins.)

I have made up my mind to make a bet on the winner for this game, but I must decide which side to take. This is essentially a classification problem–which team wins? Actually, to put this scenario in a framework agreeable for modeling, I could say that I am trying to predict the Cavaliers’ outcome w, which is 1 if they win and 0 if they lose.

There are a number of variables that I would use to help me make a prediction.

1. **Regular season team point-differential of each team.** (Point differential is the total sum of points scored by a given team minus the sum of points scored by the team’s opponent.) Actually, to account for randomness to the extent possible, it might be best to disregard point-differential in the last 6 minutes of any game, due to the possibility that scores are inflated due to “blowouts”. Additionally, it might be a good idea to express the point-differential in terms of number of possessions (via a “per-100-possessions” statistic). Moreover, one might consider only counting point-differential in games involving two-playoff teams (because these games are of “higher caliber”).

Alternatively (or additionally), **playoff team-point differential of each team** may be used. As opposed to the regular season point-differential, which has a sample size of 82, this value will undoubtedly be sensitive to small sample size. (The Cavaliers have played 12 games up to this point, and the Celtics have played 13.) Nonetheless, this value should give some indication of how well each team is playing recently, which may actually be a better independent variable than regular season point-differential.

As another alternative (or addition), **point differential of the two teams in head-to-head matchups** should be considered. Like playoff team-point differential, this value is undoubtedly sensitive to small sample bias–the Cavaliers and Celtics have played 5 games against one another this season, including Game 1 of this series– but this value may be more indicative of relative team strength because it more directly accounts for matchup-specific team strengths and weaknesses.

1. **Historical win-loss percentage of teams in the Conference Finals in a 1-0 series where the home team won the first game.** The win-percentage of teams in this exact situation should, theoretically, be indicative of how teams emotionally respond to the outcome of Game 1. By looking at only previous Conference Finals games, the relative strength of the teams is (arguably) accounted for.
2. **Injury/health status of team rosters.** This might be expressed as percentage ratio of the sum of the per-player average minutes played by “at-the-moment” healthy players (on a given team) throughout the regular season to the per-player average minutes played by the same set of players. The chances of a team winning without a full healthy roster of players decreases (in most cases).
3. **Playoff experience of team rosters.** This may be expressed as the total sum of minutes played by all individual players on a team in playoff games. It is generally agreed upon that having more playoff experience than one’s opponent can be helpful, for anecdotal reasons, such as “knowing how to handle high-pressure situations” better.
4. **Indicator of home court advantage.** This is simply a binary 1 or 0 indicated that the Celtics are playing the game at home. While this may seem very simplistic, home advantage has been shown to be a very powerful predictor throughout sports research.

## Code Setup (Pre-Question 2.2 and Beyond)

As an experienced programmer, I believe it is a good idea to define functions in order to abstract details of implementation and focus attention on results, even if the functions are used no more than once or twice. Given this mindset, I have defined lots of functions to be used for answering the assignment prompts. To help orient the avid reader, I describe these functions in some detail (although I hope that the code is sufficiently explicit/expressive to understand without additional comment).

library("tidyverse")  
library("kernlab")  
library("kknn")  
library("modelr")  
  
config <-  
 list(  
 path\_data = file.path("data", "credit\_card\_data-headers.txt"),  
 var\_y = "R1",  
 cols\_fct = c("A1", "A9", "A10", "A12", "R1"),  
 seed\_default = 42,  
 split\_trntst\_default = 0.8,  
 split\_trnvaltst\_default = c(0.5, 0.25, 0.25),  
 k\_default = 7,  
 cost\_default = 1,  
 costs\_default = 10 ^ seq(-3, 1, 1),  
 ks\_default = 2 ^ seq(1, 8, 1),  
 folds\_default = 10  
 )

Here, I’m defining some “general” functions that are used in multiple places.

* get\_var\_y() identifies the dependent variable given a formula.
* get\_sample\_idx\_random() generated a randomly-sampled vector of integers corresponding to split percentage of rows in data. The output from this function can be used to split data into training and testing splits.
* select\_y() extract the column corresponding to the dependent variable in data as a single-column matrix. (Data.frames and vectors can sometimes be problematic for modeling functions from packages problems.)
* compute\_class\_acc() calculated the classification accuracy of a model given the predictions (preds) and the actual values (actual). ~~Note that data must also be specified so that the average can be calculated indirectly via sum() and nrow(). This is done as a safe-guard against mean() throwing an error if there is a difference in the length of preds and actual (which may be the case if there is an NA in the predictions (possibly due to some kind of singularity error).~~
* get\_data\_folds() is a custom version of the crossv\_kfold() function from the {modelr} package used for k-fold cross validation.
* summarise\_acc\_at() is a function used to summarize the output from performing cross-validation and/or parameter tuning and/or testing-training, where there is more than one acc record returned.
* theme\_base() and labs\_base() are custom ggplot2::theme() and ggplot2::labs() lists.

get\_var\_y <-  
 function(formula = NULL) {  
 stopifnot(purrr::is\_formula(formula))  
 all.vars(formula)[1]  
 }  
  
get\_sample\_idx\_random <-  
 function(data = NULL,  
 split = config$split\_trntst\_default,  
 seed = config$seed\_default,  
 formula = NULL) {  
 set.seed(seed)  
 sample(seq\_len(nrow(data)), size = floor(split \* nrow(data)))  
 # Or...  
 # var\_y <- get\_var\_y(formula)  
 # caret::createDataPartition(data[, var\_y], p = split, list = FALSE)  
 }  
  
select\_y <-  
 function(data = NULL, formula = NULL, var\_y = get\_var\_y(formula = formula)) {  
 as.matrix(data[, var\_y])  
 }  
  
compute\_class\_acc <-  
 function(preds = NULL,  
 actual = NULL) {  
 acc <- mean(preds == actual)  
 acc  
 }  
  
get\_data\_folds <-  
 function(data = NULL, folds = NULL, seed = NULL) {  
 set.seed(seed)  
 data %>%  
 modelr::crossv\_kfold(k = folds, id = "fold") %>%  
 mutate\_at(vars(fold), funs(as.integer))  
 }  
  
summarise\_acc\_at <-  
 function(data = NULL, col = "acc", cols\_grp = NULL) {  
  
 ret <-  
 data %>%  
 group\_by(!!!rlang::syms(cols\_grp)) %>%  
 summarise\_at(vars(!!rlang::sym(col)), funs(n = n(), mean, min, max)) %>%  
 ungroup() %>%  
 mutate(rnk = dense\_rank(desc(mean))) %>%  
 arrange(desc(mean), desc(min), max)  
 if(max(ret$n) == 1) {  
 ret <-  
 ret %>%  
 select(!!!rlang::syms(cols\_grp), acc = mean, rnk)  
 }  
 ret  
 }  
  
theme\_base <-  
 function() {  
 theme\_minimal() +  
 theme(legend.position = "none", legend.title = element\_blank())  
 }  
  
labs\_base <-  
 function() {  
 labs(x = NULL, y = NULL)  
 }

These are functions that are used for kvsm calculations.

* The core function fit\_kvsm() is essentially a wrapper for kernlab::ksvm(), with custom default defined.
* compute\_kvsm\_a/a0() are convenience functions for extracting the weights of a fitted kvsm model.
* compute\_class\_acc\_kvsm() calculates classification accuracy given a fitted model (fit) and the data set (data) against which to check predictions. (This data may or may not be different that what was used to fit the model, depending on whether the training or testing (or validation) set accuracy is being calculated.) Internally, this function calls predict() (to generate predictions), followed by the compute\_class\_acc() function defined beforehand. The formula used to fit the model must also be provided so that the dependent variable in data can be identified. (I believe this is a better way of abstracting away details from the user because the dependent variable does not have to be known by the user.)
* do\_compute\_class\_acc\_ksvm() is a wrapper for fit\_kvsm followed by compute\_class\_acc\_kvsm().
* do\_compute\_class\_acc\_ksvm\_tune() is a wrapper for do\_compute\_class\_acc\_ksvm(), adding the functionality to specify more than one cost parameter to test (via costs).
* do\_compute\_class\_acc\_ksvm\_cv\_tune() is also a wrapper for do\_compute\_class\_acc\_ksvm(), adding two functionalities: (1) to specify multiple costs; and (2) to specify folds for k-fold cross validation.

fit\_kvsm <-  
 function(data = NULL,  
 formula = NULL,  
 cost = config$cost\_default,  
 type = "C-svc",  
 kernel = "vanilladot",  
 scale = TRUE,  
 seed = config$seed\_default) {  
 set.seed(seed)  
 kernlab::ksvm(  
 formula,  
 data = data,  
 C = cost,  
 type = type,  
 kernel = kernel,  
 scaled = scale  
 )  
 }  
  
# fit\_kvsm\_alt <-  
# function(data = NULL,  
# formula = NULL,  
# cost = config$cost\_default,  
# type = "C-classification",  
# kernel = "linear",  
# scale = TRUE,  
# seed = config$seed\_default) {  
# set.seed(seed)  
# e1071::svm(  
# formula,  
# data = data,  
# cost = cost,  
# type = type,  
# kernel = kernel,  
# scale = scale  
# )  
# }  
  
compute\_kvsm\_a <-  
 function(fit = NULL) {  
 colSums(fit@xmatrix[[1]] \* fit@coef[[1]])  
 }  
  
compute\_kvsm\_a0 <-  
 function(fit = NULL) {  
 -fit@b  
 }  
  
compute\_class\_acc\_ksvm <-  
 function(fit = NULL,  
 data = NULL,  
 formula = NULL,  
 var\_y = get\_var\_y(formula = formula)) {  
 preds <- predict(fit, data)  
 actual <- select\_y(data = data, var\_y = var\_y)  
 acc <- compute\_class\_acc(preds = preds, actual = actual)  
 acc  
 }  
  
do\_compute\_class\_acc\_ksvm <-  
 function(formula = NULL, data = NULL, newdata = data, ...) {  
 fit <- fit\_kvsm(data = data, formula = formula, ...)  
 acc <- compute\_class\_acc\_ksvm(fit = fit, data = newdata, formula = formula)  
 acc  
 }  
  
do\_compute\_class\_acc\_ksvm\_tune <-  
 function(formula = NULL,  
 data = NULL,  
 newdata = NULL,  
 seed = config$seed\_default,  
 costs = config$costs\_default,  
 type = "C-svc",  
 kernel = "vanilladot",  
 scale = TRUE,  
 ...) {  
  
 fits <-  
 tibble(cost = costs) %>%  
 mutate(acc = purrr::map(  
 cost,  
 ~ do\_compute\_class\_acc\_ksvm(  
 formula = formula,  
 data = data,  
 newdata = newdata,  
 seed = seed,  
 cost = .x,  
 type = type,  
 kernel = kernel,  
 scale = scale  
 )  
 )  
 )  
  
 accs <-  
 fits %>%  
 unnest(acc, .drop = TRUE)  
 accs  
 }  
  
do\_compute\_class\_acc\_ksvm\_cv\_tune <-  
 function(formula = NULL,  
 data = NULL,  
 folds = config$folds\_default,  
 seed = config$seed\_default,  
 costs = config$costs\_default,  
 type = "C-svc",  
 kernel = "vanilladot",  
 scale = TRUE,  
 ...) {  
  
 grid\_folds <- expand.grid(cost = costs, fold = seq(1, folds))  
 data\_folds <-  
 data %>%  
 get\_data\_folds(folds = folds, seed = seed) %>%  
 left\_join(grid\_folds, by = "fold")  
 fits <-  
 data\_folds %>%  
 mutate(acc = purrr::pmap(  
 list(  
 train,  
 test,  
 cost  
 ),  
 ~ do\_compute\_class\_acc\_ksvm(  
 formula = formula,  
 data = as.data.frame(..1),  
 newdata = as.data.frame(..2),  
 seed = seed,  
 cost = ..3,  
 type = type,  
 kernel = kernel,  
 scale = scale  
 )  
 )  
 )  
  
 accs <-  
 fits %>%  
 unnest(acc, .drop = TRUE)  
 accs  
 }

These are the kknn functions that are used.

With the notable exception of the \_loop() function, for the most part, these are analogous the kvsm function for working with kknn::kknn, with mostly the same input parameters as the kvsm set of functions.

* do\_predict\_kknn\_loop() is used for fitting a kknn() model and making predictions when not defining an explicitly test set. A loop for performing something akin to leave-one-out-cross-validation (LOOCV) is necessary in order to prevent the kernlab::kknn() algorithm from perfect classification by including individual observations among the training observations for fitting a model for that observation. The compute\_class\_acc\_kknn\_loop() and do\_compute\_class\_acc\_kknn\_loop() are analogous to the similarly named kvsm functions, with slightly different input formats. The arguments are different in order to accommodate the output from do\_predict\_kknn\_loop(), which is a vector of predictions, not a fitted model.
* fit\_kknn() is the core function here, like fit\_kvsm() is for the kvsm functions. It mainly serves as a wrapper to kknn::kknn(), with custom defaults defined. The major difference between the two fit\_\*() functions is that this one provides multiple options for calling the main modelling function (kknn().

1. The user may not define train, test, or idx\_trn, and the input data will be split randomly into two sets using the split percentage.
2. The user may specify train and test sets directly, which will cause data to be ignored.
3. The user may specify an index vector (idx\_trn) by which to subset data into training and testing splits.

In each of these possibilities, note that the input data is split into two sets, which is necessary for working with kknn::kknn(). This is not the case with kernlab::kvsm(). It’s important that the train and test arguments to kknn::kknn() represent distinct data sets in order to achieve a non-perfect accuracy of 1. (This function and its companion function could actually be used in place of the \_loop() functions described above, but the implementation is not straightforward.)

Additionally, in contrast to all other functions, this function returns a list containing the model (fit) and the test and train data that is used to fit the model. (These are not known to the user in cases 1 and 3 described above.) This functionality is implemented in order to allow an accuracy computation function (compute\_class\_acc\_kknn()) to be defined.

* compute\_class\_acc\_kknn() is analogous to compute\_class\_acc\_kvsm(). Notably, the input fit should be the fit element in the output of the fit\_kknn() function. (This is slightly different from the designed interaction of the analogous kvsm functions, where the sole output of fit\_kvsm() is a fitted model–not a list–which is then passed to compute\_class\_acc\_kvsm().)
* do\_compute\_class\_acc\_kknn(),
* do\_compute\_class\_acc\_kknn\_tune(), and do\_compute\_class\_acc\_kknn\_cv\_tune() are exactly analogous to their kvsm counterparts, in terms of functionality. The only notable difference is with the tuning functions where multiple ks may be defined (instead of costs, which is only relevant for kvsm).
* There is an additional do\_compute\_class\_acc\_kknn\_cv() which does not have a direct kvsm analogue. This kknn function is defined solely for the purpose of answering the question regarding cross-validation using kknn::kknn(), without parameter tuning. (It works identically to do\_compute\_class\_acc\_kknn\_cv\_tune(), but without the functionality for tuning k.)

do\_predict\_kknn\_loop <-  
 function(data = NULL,  
 formula = NULL,  
 k = config$k\_default,  
 distance = 2,  
 kernel = "optimal",  
 scale = TRUE,  
 seed = config$seed\_default) {  
 # NOTE: I don't think setting the seed here will make anything different.  
 set.seed(seed)  
 i <- 1  
 # preds <- rep(0, nrow(data))  
 preds <- vector("integer", nrow(data))  
 while(i <= nrow(data)) {  
 fit <-  
 kknn::kknn(  
 formula,  
 train = data[-i, ],  
 test = data[i, ],  
 k = k,  
 kernel = kernel,  
 scale = scale  
 )  
 # preds[[i]] <- as.integer(fit$fitted.values)  
 preds[[i]] <- as.integer(fit$fitted.values) - 1  
 i <- i + 1  
 }  
 preds  
 }  
  
compute\_class\_acc\_kknn\_loop <-  
 function(preds = NULL,  
 data = NULL,  
 formula = NULL,  
 var\_y = get\_var\_y(formula = formula)) {  
 actual <- select\_y(data = data, var\_y = var\_y)  
 acc <-  
 compute\_class\_acc(preds = preds, actual = actual)  
 acc  
 }  
  
do\_compute\_class\_acc\_kknn\_loop <-  
 function(formula = NULL, data = NULL, ...) {  
 preds <- do\_predict\_kknn\_loop(data = data, formula = formula, ...)  
 acc <-  
 compute\_class\_acc\_kknn\_loop(preds = preds, data = data, formula = formula)  
 acc  
 }  
  
fit\_kknn <-  
 function(data = NULL,  
 formula = NULL,  
 k = config$k\_default,  
 distance = 2,  
 kernel = "optimal",  
 scale = TRUE,  
 seed = config$seed\_default,  
 train,  
 test,  
 idx\_trn,  
 split = config$split\_trntst\_default) {  
  
 if (missing(train) | missing(test)) {  
 if (missing(idx\_trn)) {  
 idx\_trn <-  
 get\_sample\_idx\_random(data = data, split = split, seed = seed, formula = formula)  
 }  
 train <- data[idx\_trn, ]  
 test <- data[-idx\_trn, ]  
 }  
 set.seed(seed)  
 fit <-  
 kknn::kknn(  
 formula,  
 train = train,  
 test = test,  
 k = k,  
 kernel = kernel,  
 scale = scale  
 )  
 list(fit = fit,  
 train = train,  
 test = test)  
 }  
  
compute\_class\_acc\_kknn <-  
 function(fit = NULL,  
 data = NULL,  
 formula = NULL,  
 var\_y = get\_var\_y(formula = formula)) {  
 # preds <- round(fit$fitted.values)  
 preds <- fit$fitted.values  
 actual <- select\_y(data = data, var\_y = var\_y)  
 acc <-  
 compute\_class\_acc(preds = preds,  
 actual = actual)  
 acc  
 }  
  
do\_compute\_class\_acc\_kknn <-  
 function(formula = NULL, data = NULL, ...) {  
 fit <- fit\_kknn(data = data, formula = formula, ...)  
 acc <-  
 compute\_class\_acc\_kknn(fit = fit$fit,  
 data = fit$test,  
 formula = formula)  
 acc  
 }  
  
  
do\_compute\_class\_acc\_kknn\_cv <-  
 function(formula = NULL,  
 data = NULL,  
 folds = config$folds\_default,  
 seed = config$seed\_default,  
 k = config$k\_default,  
 distance = 2,  
 kernel = "optimal",  
 scale = TRUE,  
 ...) {  
  
 data\_folds <-  
 data %>%  
 get\_data\_folds(folds = folds, seed = seed)  
 fits <-  
 data\_folds %>%  
 mutate(acc = purrr::map2(  
 train,  
 test,  
 ~ do\_compute\_class\_acc\_kknn(  
 formula = formula,  
 data = data,  
 train = as.data.frame(.x),  
 test = as.data.frame(.y),  
 k = k,  
 seed = seed,  
 distance = distance,  
 kernel = kernel,  
 scale = scale  
 )  
 )  
 )  
  
 accs <-  
 fits %>%  
 unnest(acc, .drop = TRUE)  
 accs  
 }  
  
do\_compute\_class\_acc\_kknn\_tune <-  
 function(formula = NULL,  
 data = NULL,  
 ks = config$ks\_default,  
 seed = config$seed\_default,  
 distance = 2,  
 kernel = "optimal",  
 scale = TRUE,  
 train,  
 test,  
 idx\_trn,  
 split = config$split\_trntst\_default,  
 ...) {  
 grid <- tibble(k = as.integer(ks))  
 if (missing(train) | missing(test)) {  
 if (missing(idx\_trn)) {  
 idx\_trn <-  
 get\_sample\_idx\_random(data = data, split = split, seed = seed, formula = formula)  
 }  
 train <- data[idx\_trn, ]  
 test <- data[-idx\_trn, ]  
 }  
 fits <-  
 grid %>%  
 mutate(acc = purrr::map(  
 k,  
 ~ do\_compute\_class\_acc\_kknn(  
 formula = formula,  
 data = data,  
 train = train,  
 test = test,  
 seed = seed,  
 k = .x,  
 distance = distance,  
 kernel = kernel,  
 scale = scale  
 )  
 ))  
  
 accs <-  
 fits %>%  
 unnest(acc, .drop = TRUE)  
 accs  
 }  
  
do\_compute\_class\_acc\_kknn\_cv\_tune <-  
 function(formula = NULL,  
 data = NULL,  
 folds = config$folds\_default,  
 seed = config$seed\_default,  
 ks = config$ks\_default,  
 distance = 2,  
 kernel = "optimal",  
 scale = TRUE,  
 ...,  
 return\_details = FALSE) {  
 grid\_folds <- expand.grid(k = as.integer(ks), fold = seq(1, folds))  
 data\_folds <-  
 data %>%  
 get\_data\_folds(folds = folds, seed = seed) %>%  
 left\_join(grid\_folds, by = "fold")  
 fits <-  
 data\_folds %>%  
 mutate(acc = purrr::pmap(  
 list(  
 train,  
 test,  
 k  
 ),  
 ~ do\_compute\_class\_acc\_kknn(  
 formula = formula,  
 data = data,  
 train = as.data.frame(..1),  
 test = as.data.frame(..2),  
 seed = seed,  
 k = ..3,  
 distance = distance,  
 kernel = kernel,  
 scale = scale  
 )  
 )  
 )  
  
 accs <-  
 fits %>%  
 unnest(acc, .drop = TRUE)  
 accs  
 }

## Question 2.2.1

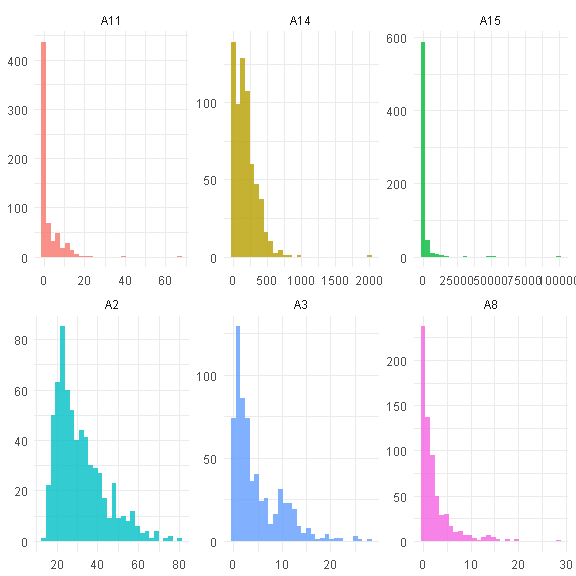
To begin, I first need to import the data. Note that I explicitly convert the response variable to a factor. This is particularly useful for the kknn models created later.

data <-  
 config$path\_data %>%  
 readr::read\_delim(delim = "\t") %>%   
 mutate\_at(vars(c(config$var\_y)), funs(factor))  
data

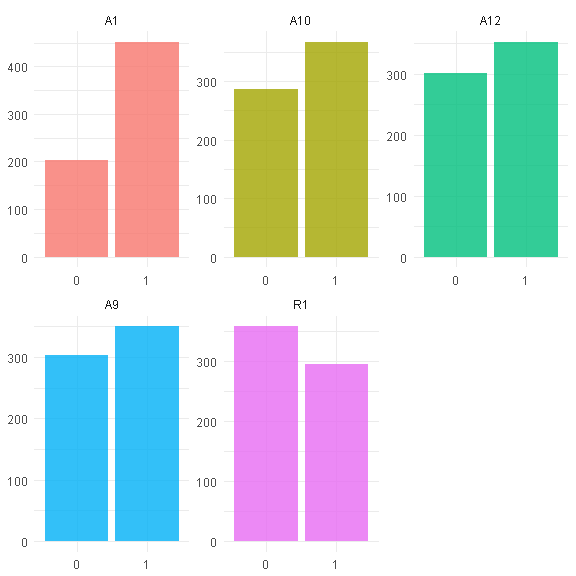
## # A tibble: 654 x 11  
## A1 A2 A3 A8 A9 A10 A11 A12 A14 A15 R1   
## <int> <dbl> <dbl> <dbl> <int> <int> <int> <int> <int> <int> <fct>  
## 1 1 30.83 0 1.250 1 0 1 1 202 0 1   
## 2 0 58.67 4.460 3.040 1 0 6 1 43 560 1   
## 3 0 24.50 0.5000 1.500 1 1 0 1 280 824 1   
## 4 1 27.83 1.540 3.750 1 0 5 0 100 3 1   
## 5 1 20.17 5.625 1.710 1 1 0 1 120 0 1   
## 6 1 32.08 4.000 2.500 1 1 0 0 360 0 1   
## 7 1 33.17 1.040 6.500 1 1 0 0 164 31285 1   
## 8 0 22.92 11.58 0.04000 1 1 0 1 80 1349 1   
## 9 1 54.42 0.5000 3.960 1 1 0 1 180 314 1   
## 10 1 42.50 4.915 3.165 1 1 0 0 52 1442 1   
## 11 1 22.08 0.8300 2.165 0 1 0 0 128 0 1   
## 12 1 29.92 1.835 4.335 1 1 0 1 260 200 1   
## 13 0 38.25 6.000 1.000 1 1 0 0 0 0 1   
## 14 1 48.08 6.040 0.04000 0 1 0 1 0 2690 1   
## 15 0 45.83 10.50 5.000 1 0 7 0 0 0 1   
## 16 1 36.67 4.415 0.2500 1 0 10 0 320 0 1   
## 17 1 28.25 0.8750 0.9600 1 0 3 0 396 0 1   
## 18 0 23.25 5.875 3.170 1 0 10 1 120 245 1   
## 19 1 21.83 0.2500 0.6650 1 1 0 0 0 0 1   
## 20 0 19.17 8.585 0.7500 1 0 7 1 96 0 1   
## # ... with 634 more rows

Let’s quickly inspect the data with some histograms. Note that I convert all categorical variables to factors here because ggplot2::geom\_col() should be used instead of ggplot2::geom\_histogram() for categorical variables.

data\_fct <-  
 data %>%  
 mutate\_at(vars(c(config$cols\_fct)), funs(factor))  
  
viz\_cnt\_num <-  
 data\_fct %>%   
 select\_if(is.numeric) %>%   
 gather(col, value) %>%   
 ggplot(aes(x = value, fill = col)) +  
 geom\_histogram(alpha = 0.8, bins = 30) +  
 facet\_wrap( ~ col, scales = "free") +  
 theme\_base() +  
 labs\_base()  
viz\_cnt\_num



viz\_cnt\_fct <-  
 data\_fct %>%   
 select\_if(is.factor) %>%   
 gather(col, value) %>%   
 ggplot(aes(x = value, fill = col)) +  
 geom\_bar(alpha = 0.8) +  
 facet\_wrap( ~ col, scales = "free") +  
 theme\_base() +  
 labs\_base()  
viz\_cnt\_fct



First, I will look at a model fitted using all possible predictors.

vars\_x <- setdiff(names(data), config$var\_y)  
fmla\_chr <-  
 paste0(config$var\_y, " ~ ", paste(vars\_x, collapse = " + "))  
fmla\_chr

## [1] "R1 ~ A1 + A2 + A3 + A8 + A9 + A10 + A11 + A12 + A14 + A15"

I’ll fit the model, first using a cost () of 1 (which is how the fit\_kvsm() function is defined above). If my interpretation of support vector machines (SVMs) is correct, I believe that this means that the classifier may allow up to 1 observations to be on the “wrong” side of the classifying hyperplane. (p. 347, *ISLR*)

" For > 0 no more than observations can be on the wrong side of the hyperplane… As the budget increases, we become more tolerant of violations to the margin, and so the margin will widen. Conversely, as decreases, we become less tolerant of violations to the margin and so the margin narrows… controls the bias-variance trade-off of the statistical learning technique. When is small, we seek narrow margins that are rarely violated; this amounts to a classifier that is highly fit to the data, which may have low bias but high variance. On the other hand, when C is larger, the margin is wider and we allow more violations to it; this amounts to fitting the data less hard and obtaining a classifier that is potentially more biased but may have lower variance."

fmla <- formula(fmla\_chr)  
  
model\_ksvm <-  
 fit\_kvsm(  
 data = data,   
 formula = fmla  
 )

## Setting default kernel parameters

model\_ksvm

## Support Vector Machine object of class "ksvm"   
##   
## SV type: C-svc (classification)   
## parameter : cost C = 1   
##   
## Linear (vanilla) kernel function.   
##   
## Number of Support Vectors : 190   
##   
## Objective Function Value : -179.385   
## Training error : 0.136086

Note that this cost value returns a reasonable distribution of 0s and 1s for the dependent variable given the actual distribution of this variable.

predict(model\_ksvm, data) %>% table()

## .  
## 0 1   
## 303 351

select\_y(data = data, var\_y = config$var\_y) %>% table()

## .  
## 0 1   
## 358 296

kvsm\_cmat <-  
 bind\_cols(  
 actual = select\_y(data = data, var\_y = config$var\_y),  
 predicted = predict(model\_ksvm, data)  
 ) %>%   
 table()  
kvsm\_cmat

## predicted  
## actual 0 1  
## 0 286 72  
## 1 17 279

The coefficients of this ksvm model can calculated from components of the model object returned by kernlab::ksvm(). (This is shown in the function definitions above.) Note that the weights a are equivalent to the inner product of all observations used for training (and is actually only non-zero for the “support vector” points themselves). a0 represents the intercept term.

a <-  
 compute\_kvsm\_a(fit = model\_ksvm) %>%   
 tibble::enframe()  
a0 <- compute\_kvsm\_a0(fit = model\_ksvm)  
a

## # A tibble: 10 x 2  
## name value  
## <chr> <dbl>  
## 1 A1 -0.001103   
## 2 A2 -0.0008981  
## 3 A3 -0.001607   
## 4 A8 0.002904   
## 5 A9 1.005   
## 6 A10 -0.002985   
## 7 A11 -0.0002035  
## 8 A12 -0.0005505  
## 9 A14 -0.001252   
## 10 A15 0.1064

a0

## [1] 0.08148382

Additionally, the accuracy can be calculated.

acc\_ksvm <-  
 compute\_class\_acc\_ksvm(  
 fit = model\_ksvm,   
 data = data,   
 formula = fmla  
 )  
acc\_ksvm

## [1] 0.8639144

# Or, directly, without the model fit.  
acc\_ksvm <-  
 do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data  
 )

## Setting default kernel parameters

acc\_ksvm

## [1] 0.8639144

To convince the reader that a cost of 1 is reasonable, note that model accuracy decreases using the same formula for very large and small values of cost.

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 cost = 1e3  
)

## Setting default kernel parameters

## [1] 0.8623853

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 cost = 1e-5  
)

## Setting default kernel parameters

## [1] 0.5474006

Interestingly, note that the only change that occurs in model accuracy when increasing values of cost beyond 1 (approximately) is for the value equivalent to the number of observations in the data set. This “threshold” describes the “breaking point” at which all observation may be misclassified.

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 cost = nrow(data) - 1  
)

## Setting default kernel parameters

## [1] 0.8623853

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 cost = nrow(data)  
)

## Setting default kernel parameters

## [1] 0.8639144

Additionally, note that if I use less predictors, the accuracy of the model decreases (holding everything else constant with the model specification).

do\_compute\_class\_acc\_ksvm(  
 formula = formula("R1 ~ A14 + A15"),  
 data = data  
)

## Setting default kernel parameters

## [1] 0.648318

With these changes to model parameters investigated (for formula and cost), I think it is fair to say that my first choice of model–including all predictors and setting cost = 1–is agreeable.

## Question 2.2.2

Here, I’ll calculate the classification accuracy of kernlab::kvsm() with some kernels other than "vanilla dot" (holding everything else constant). Note that the accuracy of the model fit with "rbfdot" (a radial kernel) is slightly better than that with "vanilla dot" (a linear kernel), while that fit with "polydot" (a polynomial kernel) is equal and that fit with "tanhdot" (a kernel based on the tanh trigonometric transformation) is less (given the seed that I have chosen).

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 kernel = "rbfdot"  
)

## [1] 0.8715596

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 kernel = "polydot"  
)

## Setting default kernel parameters

## [1] 0.8639144

do\_compute\_class\_acc\_ksvm(  
 formula = fmla,  
 data = data,  
 kernel = "tanhdot"  
)

## Setting default kernel parameters

## [1] 0.7217125

## Question 2.2.3

Now I will evaluate a model fit with the same data set and the same formula, using kknn::kknn().

(Note the following:

* A default value of 7 is used for k, as defined above in fit\_kknn().
* Because we are not defining an explicitly test set here, we must “manually” define one in a loop including all observations not including one observation (i), fit a model for each observation i separately, and make a single prediction for each i. (This is the purpose of my \_kknn\_loop() functions.)

preds\_kknn\_loop <-  
 do\_predict\_kknn\_loop(  
 data = data,  
 formula = fmla  
 )  
preds\_kknn\_loop

## [1] 1 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1  
## [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 0 0 1 1 1 0 1 1 1 1 1 1 1  
## [71] 0 0 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1  
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1  
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 1 1 1  
## [176] 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1  
## [ reached getOption("max.print") -- omitted 454 entries ]

acc\_kknn\_loop <-  
 compute\_class\_acc\_kknn\_loop(  
 preds = preds\_kknn\_loop,  
 data = data,  
 var\_y = config$var\_y  
 )  
acc\_kknn\_loop

## [1] 0.8470948

# Or, directly, without the model fit.  
acc\_kknn\_loop <-  
 do\_compute\_class\_acc\_kknn\_loop(  
 formula = fmla,  
 data = data\_fct  
 )  
acc\_kknn\_loop

## [1] 0.8470948

For pedagogical purposes, I’ll now demonstrate the creation and usage of a kknn model using a more traditional train-test approach. I’ll use the “non-loop” functions that I’ve defined before. Also, I’ll coerce the response variable to a factor so that kknn::kknn() always predicts either a 0 or 1 and not some value in between.

(Note that the the split percentage used to define the required train and test arguments of kknn::kknn() has been assigned a default value of 0.8.)

model\_kknn <- fit\_kknn(data = data, formula = fmla)  
model\_kknn$fit

##   
## Call:  
## kknn::kknn(formula = formula, train = train, test = test, k = k, kernel = kernel, scale = scale)  
##   
## Response: "nominal"

We can see the components of the returned object, including the class of the dependent variable (response), the fitted values (fitted.values), and the probabilities (probs). Note that kknn::kknn() recognizes that the dependent variable is a factor, and that the distribution of 0s and 1s among the fitted values seems reasonable.

model\_kknn$fit$response

## [1] "nominal"

model\_kknn$fit$fitted.values %>% head()

## [1] 0 1 1 1 1 1  
## Levels: 0 1

model\_kknn$fit$prob %>% head()

## 0 1  
## [1,] 0.7864444 0.2135556  
## [2,] 0.0000000 1.0000000  
## [3,] 0.2008116 0.7991884  
## [4,] 0.0127440 0.9872560  
## [5,] 0.2330221 0.7669779  
## [6,] 0.0000000 1.0000000

table(model\_kknn$fit$fitted.values)

##   
## 0 1   
## 75 56

I’ll go ahead and calculate the model’s classification accuracy.

acc\_kknn <-  
 compute\_class\_acc\_kknn(  
 fit = model\_kknn$fit,   
 data = model\_kknn$test,   
 formula = fmla  
 )  
acc\_kknn

## [1] 0.870229

# Or, directly, without the model fit.  
acc\_kknn <-  
 do\_compute\_class\_acc\_kknn(  
 formula = fmla,  
 data = data\_fct  
 )  
acc\_kknn

## [1] 0.870229

Now, I’ll try to identify a good value of k. (Specifically, I’ll try values of 2, 4, 8, 16, 32, 64, 128, 256, which have been defined as the default value in the function.)

acc\_kknn\_tune <-  
 do\_compute\_class\_acc\_kknn\_tune(  
 formula = fmla,  
 data = data  
 )  
acc\_kknn\_tune %>%   
 summarise\_acc\_at(cols\_grp = "k")

## # A tibble: 8 x 3  
## k acc rnk  
## <int> <dbl> <int>  
## 1 128 0.9008 1  
## 2 256 0.8855 2  
## 3 32 0.8779 3  
## 4 64 0.8779 3  
## 5 8 0.8702 4  
## 6 16 0.8702 4  
## 7 2 0.8168 5  
## 8 4 0.8168 5

While the accuracy is highest for k = 128, note that the accuracy does not improve significantly with increasing values of k beyond 8. And, given that these accuracy values are not calculated using a final “hold-out” set–internally, the function splits the data into a training and test set (and then calculates the accuracy against the test set), but this is not a “true” test set–the slightly higher values of accuracy for greater values of k are suspicious. In fact, I would say that choosing a large value of k would be **overfitting**. That is, I could maximize the accuracy given this data set by choosing a large value of k, but the accuracy might not be as high (as that of another model using a lower value of k) given a new set of unseen records from the original data source.

With all of this in mind, I think a good value of k is 8, given that the model is fit using all predictors.

## Question 3.1.a

Finding the “optimal” k value for my kknn model is certainly important. However, the importance of estimating the expected accuracy of such a model should not be overlooked. Calculating a robust estimate of classification accuracy can be done with k-fold cross-validation. Here, I’ll implement 10 cross-validation for a kknn model with k = 8. (Note that a value of 10 is used by default for the number of folds for cross validation for the do\_compute\_class\_acc\_kknn\_cv() function.

acc\_kknn\_cv <-  
 do\_compute\_class\_acc\_kknn\_cv(  
 formula = fmla,  
 data = data,  
 k = 8  
 )  
acc\_kknn\_cv

## # A tibble: 10 x 2  
## fold acc  
## <int> <dbl>  
## 1 1 0.8333  
## 2 2 0.8788  
## 3 3 0.7879  
## 4 4 0.8485  
## 5 5 0.8462  
## 6 6 0.8462  
## 7 7 0.8000  
## 8 8 0.9231  
## 9 9 0.8462  
## 10 10 0.8769

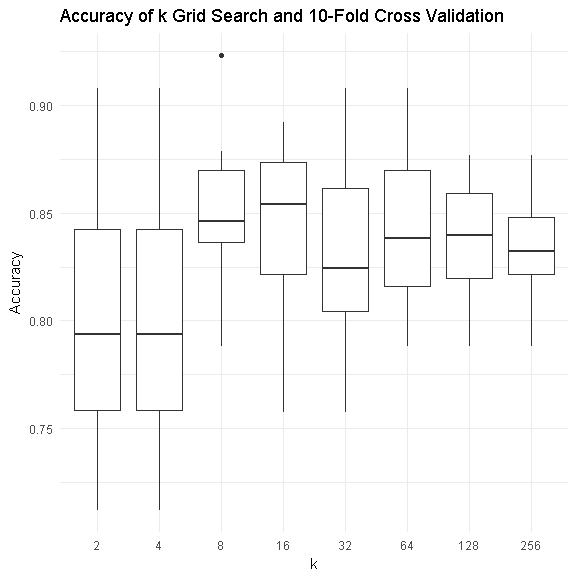
And, because I’m curious, what would the cross validation accuracies be for different values of k?

acc\_kknn\_cv\_tune <-  
 do\_compute\_class\_acc\_kknn\_cv\_tune(  
 formula = fmla,  
 data = data  
 )  
summ\_acc\_kknn\_cv\_tune <-  
 acc\_kknn\_cv\_tune %>%  
 summarise\_acc\_at(cols\_grp = "k")  
summ\_acc\_kknn\_cv\_tune

## # A tibble: 8 x 6  
## k n mean min max rnk  
## <int> <int> <dbl> <dbl> <dbl> <int>  
## 1 8 10 0.8487 0.7879 0.9231 1  
## 2 16 10 0.8410 0.7576 0.8923 2  
## 3 64 10 0.8410 0.7879 0.9077 3  
## 4 128 10 0.8380 0.7879 0.8769 4  
## 5 256 10 0.8349 0.7879 0.8769 5  
## 6 32 10 0.8288 0.7576 0.9077 6  
## 7 2 10 0.7983 0.7121 0.9077 7  
## 8 4 10 0.7983 0.7121 0.9077 7

Let’s visualize these results.

viz\_acc\_kknn\_cv\_tune <-  
 acc\_kknn\_cv\_tune %>%  
 mutate\_at(vars(k), funs(forcats::fct\_reorder(factor(.), k))) %>%  
 ggplot(aes(x = k, y = acc)) +  
 geom\_boxplot() +  
 theme\_base() +  
 labs(  
 title = str\_wrap("Accuracy of k Grid Search and 10-Fold Cross Validation", 80),  
 x = "k",  
 y = "Accuracy"  
 )  
viz\_acc\_kknn\_cv\_tune



The results here seem to indicate that my choice of k = 8 for the kknn model is fairly optimal.

Now, stepping back from kknn for moment, it’s straightforward to perform cross-validation with parameter tuning for ksvm model using a similar approach (with the functions that have been defined above).

acc\_ksvm\_cv\_tune <-  
 do\_compute\_class\_acc\_ksvm\_cv\_tune(  
 formula = fmla,  
 data = data  
 )

## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters

summ\_acc\_ksvm\_cv\_tune <-  
 acc\_ksvm\_cv\_tune %>%  
 summarise\_acc\_at(cols\_grp = "cost")  
summ\_acc\_ksvm\_cv\_tune

## # A tibble: 5 x 6  
## cost n mean min max rnk  
## <dbl> <int> <dbl> <dbl> <dbl> <int>  
## 1 0.01000 10 0.8624 0.8030 0.9242 1  
## 2 0.1000 10 0.8624 0.8030 0.9242 1  
## 3 1.000 10 0.8624 0.8030 0.9242 1  
## 4 10.00 10 0.8624 0.8030 0.9242 1  
## 5 0.001000 10 0.8289 0.7727 0.8769 2

A reasonable value of cost seems to be 1, as was shown before without cross-validation (although there is no/very little difference in the average accuracy calculated for different cost values with this approach).

## Question 3.1.b

Finally, I’ll try to identify the “best” ksvm or kknn model using a train-validation-test approach, along with some parameter tuning (for cost with ksvm and k with kknn). I’ll create a new function here to implement the core of this technique.

do\_trnvaltst\_ksvm\_kknn <-  
 function(formula = NULL,  
 data = NULL,  
 costs = config$costs\_default,  
 ks = config$ks\_default,  
 splits = config$split\_trnvaltst\_default,  
 seed = config$seed\_default,  
 ...) {  
  
 idx\_trnval <-  
 get\_sample\_idx\_random(data = data, split = sum(splits[1:2]), seed = seed, formula = formula)  
 data\_trnval <- data[idx\_trnval, ]  
  
 split2 <- splits[1:2] / sum(splits[1:2])  
 idx\_trn <-  
 get\_sample\_idx\_random(data = data, split = split2[1], seed = seed, formula = formula)  
 data\_trn <- data[idx\_trn, ]  
 data\_val <- data[-idx\_trn, ]  
 data\_tst <- data[-idx\_trnval, ]  
  
 accs\_trn\_ksvm <-  
 do\_compute\_class\_acc\_ksvm\_tune(  
 formula = formula,  
 data = data\_trn,  
 newdata = data\_trn,  
 costs = costs  
 )  
 accs\_val\_ksvm <-  
 do\_compute\_class\_acc\_ksvm\_tune(  
 formula = formula,  
 data = data\_trn,  
 newdata = data\_val,  
 costs = costs  
 )  
 accs\_tst\_ksvm <-  
 do\_compute\_class\_acc\_ksvm\_tune(  
 formula = formula,  
 data = data\_trn,  
 newdata = data\_tst,  
 costs = costs  
 )  
 accs\_trn\_kknn <-  
 do\_compute\_class\_acc\_kknn\_tune(  
 formula = formula,  
 data = data\_trn,  
 train = data\_trn,  
 test = data\_trn,  
 ks = ks  
 )  
 accs\_val\_kknn <-  
 do\_compute\_class\_acc\_kknn\_tune(  
 formula = formula,  
 data = data\_trn,  
 train = data\_trn,  
 test = data\_val,  
 ks = ks  
 )  
 accs\_tst\_kknn <-  
 do\_compute\_class\_acc\_kknn\_tune(  
 formula = formula,  
 data = data\_trn,  
 train = data\_trn,  
 test = data\_tst,  
 ks = ks  
 )  
 accs <-  
 bind\_rows(  
 accs\_trn\_ksvm %>% rename(param = cost) %>% mutate(method = "ksvm", set = "trn"),  
 accs\_val\_ksvm %>% rename(param = cost) %>% mutate(method = "ksvm", set = "val"),  
 accs\_tst\_ksvm %>% rename(param = cost) %>% mutate(method = "ksvm", set = "tst"),  
 accs\_trn\_kknn %>% rename(param = k) %>% mutate(method = "kknn", set = "trn"),  
 accs\_val\_kknn %>% rename(param = k) %>% mutate(method = "kknn", set = "val"),  
 accs\_tst\_kknn %>% rename(param = k) %>% mutate(method = "kknn", set = "tst")  
 )  
 accs  
 }

Now I call do\_trnvaltst\_ksvm\_kknn() and then analyze the results.

accs\_trnvaltst <-  
 do\_trnvaltst\_ksvm\_kknn(  
 formula = fmla,  
 data = data  
 )

## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters   
## Setting default kernel parameters

# Average accuracy using the validation set only.  
accs\_trnvaltst %>%  
 filter(set == "val") %>%  
 summarise\_acc\_at(cols\_grp = c("method", "param", "set"))

## # A tibble: 13 x 5  
## method param set acc rnk  
## <chr> <dbl> <chr> <dbl> <int>  
## 1 kknn 16.00 val 0.8807 1  
## 2 kknn 128 val 0.8807 1  
## 3 kknn 256 val 0.8761 2  
## 4 kknn 8.000 val 0.8716 3  
## 5 ksvm 0.01000 val 0.8670 4  
## 6 ksvm 0.1000 val 0.8670 4  
## 7 ksvm 1.000 val 0.8670 4  
## 8 ksvm 10.00 val 0.8670 4  
## 9 kknn 32.00 val 0.8624 5  
## 10 kknn 64.00 val 0.8624 5  
## 11 ksvm 0.001000 val 0.8349 6  
## 12 kknn 2.000 val 0.7844 7  
## 13 kknn 4.000 val 0.7844 7

# Average accuracy using the test set only.  
accs\_trnvaltst %>%  
 filter(set == "tst") %>%  
 summarise\_acc\_at(cols\_grp = c("method", "param", "set"))

## # A tibble: 13 x 5  
## method param set acc rnk  
## <chr> <dbl> <chr> <dbl> <int>  
## 1 kknn 16.00 tst 0.8963 1  
## 2 kknn 128 tst 0.8963 1  
## 3 kknn 256 tst 0.8902 2  
## 4 kknn 8.000 tst 0.8841 3  
## 5 ksvm 0.01000 tst 0.8841 3  
## 6 ksvm 0.1000 tst 0.8841 3  
## 7 ksvm 1.000 tst 0.8841 3  
## 8 ksvm 10.00 tst 0.8841 3  
## 9 kknn 32.00 tst 0.8780 4  
## 10 kknn 64.00 tst 0.8780 4  
## 11 ksvm 0.001000 tst 0.8354 5  
## 12 kknn 2.000 tst 0.7805 6  
## 13 kknn 4.000 tst 0.7805 6

Based on the validation set accuracies, I would choose a kknn model with k = 16 because it has the best accuracy. (If two models are shown to have equal accuracy, it is generally best to choose the one with a lower value of k in order to prevent overfitting.) Nonetheless, there appears to be some other models with either identical or near identical validation and test set performance. accuracy of all models among the test set. However, if I am to choose my model based on the validation set results and use the test set results to get a better estimate of my chosen model, I must stick with my choice based on the validation set, even if another model is shown to have better classification accuracy on the test set.

Note that the results found here are sensitive to a number of choices I have made, including the following.

* The value of the seed used for randomizing the index used to split the whole data set.
* The manner by which the whole data set is split. (A method that “rotates” an index by a specified modulus could have been used.).
* A different set of percentages used to split the whole data set into three sets.

In all, one should understand the importance of using different sets of data for performing different tasks when comparing more than one type of model (in this case, ksvm and kknn):

* Tuning parameters (using the training set);
* Choosing a model (using the validation set); and
* Estimating “final” accuracy (using the test set).