Predicting Kills Per Set for NESCAC 2024 Players

Using Attack Attempts Per Set, Hitting Percentage, and Position

Setting up our data

```
library(readxl)
library(tidyverse)
library(ggplot2)
```

The first step to creating an algorithm that can predict kills per set is to set up a dataset that has all of the values we need. The dataset 'nescac_stats' includes the offensive stats for all players in the NESCAC for 2024 who have played in at least one match (data is updated to September 22, 2024 stats).

```
nescac_stats <- read_excel("nescac_stats.xlsx")

nescac_stats <- mutate(nescac_stats, TAperS = TA/SP)
names(nescac_stats)[names(nescac_stats) == 'K/S'] <- 'KperS'

nescac_stats <- nescac_stats |>
  mutate(pos_clean = case_when(Position == "OH" ~ "PIN",
                               Position == "RH" ~ "PIN",
                               Position == "PIN" ~ "PIN",
                               Position == "MB" ~ "MIDDLE",
                               Position == "DS" ~ "LIBERO",
                               Position == "L" ~ "LIBERO",
                               Position == "DS/L" ~ "LIBERO",
                               TRUE ~ "SETTER"))
```

The code above imports the data, creates a new column called 'TAperS' which represents attack attempts per set, and also renames the 'K/S' column to 'KperS' in the dataset. It also creates a new column called 'pos_clean' which puts the positions of players into more general categories to help with our analysis later on.

Predicting Kills per Set based on Attempts per Set and Hitting Percentage

Before we add in the categorical variable of position, let's look at a kNN function that can use only attacks per set and hitting percentage to predict kills per set.

Here is some information on what each variable means in context:

- Kills per set: A kill is when a player gets the ball over the net in a way where the other team is unable to receive it or control it. Kills per set is the statistic representing how many kills a player gets per sets they played.
- Attempts per set: An attempt is anytime you put the ball over the net with intention to try and get a kill. Attempts per set is the statistic representing how many attempts a player gets per sets they played.
- Hitting percentage: This statistic is a representation of how well a player is hitting. This is calculated using the formula (kills - errors)/attempts.

The higher the amount of attempts a player has, the more chances they have to get kills, creating a positive (linear) relationship between attempts per set and kills per set. A higher hitting percentage also usually means a higher kills per set for a player because they are hitting efficiently and therefore having more kills. When looking just at the data between -0.5 and 0.5 hitting percentage (a normal range for hitters), we can see that this relationship is relatively positive. There are definitely some outliers in this data, which are mostly due to the position of the player which we will look at later.

Now, let's create our kNN function to predict kills per set based on these two variables.

```
kills_per_set <- function(k, TAperS_input, PCT_input, data){
  data |>
  mutate(distance = sqrt((TAperS - TAperS_input)^2
                         + ((PCT - PCT_input)*10)^2)) |>
  arrange(distance) |>
  head(k) |>
  summarize(mean(KperS))
}
```

This function takes in a k value, attacks per set value, hitting percentage value, and the data set we want to analyze. It creates a distance variable that we can use to find the closest k neighbors. When calculating the distance, we want to make sure that our two variables are on the same scale (or at least have a similar weight), so in order to do this, I multiplied the hitting percentage variable difference by 10. Since hitting percentage is on a scale of 0 to 1, and attacks per set is any positive integer usually in the range of 0 to 10, multiplying hitting percentage by 10 will give us a more useful distance variable to find the actual closest neighbors. We then find the closest k players and take the mean of their kills per set in order to get a prediction for the kills per set for our input player's stats.

Now, let's test our algorithm. Below we are creating a testing data set, and a training data set which we will use to see how good our algorithm is on a random dataset (pulled from 'nescac_stats').

```
set.seed(1)
training_rows <- sample(1:nrow(nescac_stats),
                        size = nrow(nescac_stats)/2)
train_data <- nescac_stats[training_rows, ]
test_data <- nescac_stats[-training_rows, ]
```

Let's set our k equal to 10, and for each player in our test dataset, use our training dataset find a prediction for their kills per set. Then, we will find the (absolute value) residual for each player (actual kills per set - predicted kills per set), and take the mean of all of these residuals in order to see, on average, how far off our algorithm's prediction of kills per set is compared to the actual statistic.

```
k <- 10
ans <- data.frame(KpS = 1:nrow(test_data))

for(i in 1:nrow(test_data)) {
  prediction <- kills_per_set(k,
                        test_data$TAperS[i],
                        test_data$PCT[i],
                        train_data)
  ans$KpS[i] <- as.numeric(abs(test_data$KperS[i] - prediction))
}
mean1 <- mean(ans$KpS)
mean1
```

```
[1] 0.1602245
```

Using k = 10, our algorithm will on average predict a kills per set value that is 0.160 off from the actual value. Since a usual kills per set stat for a player ranges from 0 to 4, this mean

residual of 0.16 shows that our algorithm is quite accurate. Now we will look at how using position as a third input variable affects our algorithm's predictions.

Adding the Categorical Variable, Position

In volleyball, there are technically six positions, but some positions will have similar kills stats, so we can group them into four general positions for this analysis:

- PIN: These are the players who hit the ball the most. They include outsides (OH) and right sides (RH), and should have the highest kills per set stats on the team because they are usually set the most.
- MIDDLE: Middles play in the middle of the front row and hit tempo balls, which means they are set less than the pins, which means they will have a kills per set range that is less than the pins.
- LIBERO: This position includes liberos (L) and defensive specialists (DS) in this case, beacuse both positions only play in the backrow and therefore have little to no attempts per set. It is a possibility for these players to have attempts and kills during scramble plays (they have to take the third contact) or situations where a team needs to set the back row, but their stats in these areas are usually very low.
- SETTER: The setters are the players that take the second ball and set up the pins and middles for an attack. This means that they are not usually taking attempts and getting kills. However, a setter can (and usually does at least a few times per set) send over the ball on the second contact to try and get a kill. This gives them a higher attempts and kills per set stat than liberos, but still not close to middles and pins.

Given this context, we can create a scale for position to create a way to add it into our distance calculation to find our kNN.

```
nescac_stats <- nescac_stats |>
  mutate(pos_value = case_when(pos_clean == "PIN" ~ 0.2,
                               pos_clean == "MIDDLE" ~ 0.3,
                               pos_clean == "LIBERO" ~ 0.9,
                               TRUE ~ 0.7))
```

This scale was chosen in order to create two "clusters" of positions: pins/middles and liberos/setters with slighlty differing values to account for the difference in kills per set for each position.

Now we will take the same steps we used for the first kNN algorithm, but add our position variable in when finding the distance between our input and its nearest neighbors.

```r
kills_per_set2 <- function(k, TAperS_input, PCT_input, pos_input, data){
  data |>
  mutate(distance = sqrt((TAperS - TAperS_input)^2
                         + ((PCT - PCT_input)*10)^2 + (pos_value - pos_input)^2)) |>
  arrange(distance) |>
  head(k) |>
  summarize(mean(KperS))
}
```

Again, let's test this algorithm:

```r
set.seed(1)
training_rows <- sample(1:nrow(nescac_stats),
                        size = nrow(nescac_stats)/2)
train_data <- nescac_stats[training_rows, ]
test_data <- nescac_stats[-training_rows, ]


k <- 10
ans <- data.frame(KpS = 1:nrow(test_data))
for(i in 1:nrow(test_data)) {
  prediction <- kills_per_set2(k,
                       test_data$TAperS[i],
                       test_data$PCT[i],
                       test_data$pos_value[i],
                       train_data)
  ans$KpS[i] <- as.numeric(abs(test_data$KperS[i] - prediction))
}
mean(ans$KpS)
```

```
[1] 0.1596531
```

We can see here that using k = 10, we got a mean (absolute value) residual of 0.159 for this test data set. But this leaves us with a few questions:

- What actually is the best k for our algorithm?
- In general, did adding the position variable help our algorithm's accuracy?

Finding the Best k

In order to find the best k, we want to find the value that will give us the smallest mean residual for a given data set. Let's create a dataset that will show us the mean of the residuals for k values 1 to 15, based on our test data.

```
kvalues <- data.frame(k = 1:15)

for (j in 1:15) {
  x = kvalues$k[j]
  ansTemp <- data.frame(KpS = 1:nrow(test_data))
  for(i in 1:nrow(test_data)) {
    prediction <- kills_per_set2(x,
                          test_data$TAperS[i],
                          test_data$PCT[i],
                          test_data$pos_value[i],
                          train_data)
    ansTemp$KpS[i] <- as.numeric(abs(test_data$KperS[i] - prediction))
  }

  kvalues$mean[j] <- mean(ansTemp$KpS)
}

kvalues
```

```
    k      mean
1   1 0.1818367
2   2 0.1504592
3   3 0.1480952
4   4 0.1426786
5   5 0.1367959
6   6 0.1367857
7   7 0.1406414
8   8 0.1522832
9   9 0.1519501
10 10 0.1596531
11 11 0.1548052
12 12 0.1598044
13 13 0.1646782
14 14 0.1668076
15 15 0.1674762
```

This table shows us that k = 6 is our best value for k.

Now that we have the best k value, we can find out how much our position variable actually added to our algorithm's prediction.

Looking at Our Alorithm's Accuracy Across Multiple Testing Datasets

Just picking one random data set may not give us the best results especially since our original dataset (nescac_stats) is relatively small. So, we can change the seed we use to create our testing and training datasets, then analyze the mean residual for each.

On our algorithm that uses position as a variable, let's use 20 different seeds, crate a table called 'accuracy' and then find the mean of the mean residual for each dataset.

```
accuracy <- data.frame(pos_function = 1:20)

for(i in 1:20) {

  set.seed(i)
  training_rows <- sample(1:nrow(nescac_stats),
                          size = nrow(nescac_stats)/2)
  train_data <- nescac_stats[training_rows, ]
  test_data <- nescac_stats[-training_rows, ]

  k <- 6

  ansTemp <- data.frame(KpS = 1:nrow(test_data))

  for(j in 1:nrow(test_data)) {
  prediction <- kills_per_set2(k,
                       test_data$TAperS[j],
                       test_data$PCT[j],
                       test_data$pos_value[j],
                       train_data)
  ansTemp$KpS[j] <- as.numeric(abs(test_data$KperS[j] - prediction))
  }

  accuracy$pos_function[i] <- mean(ansTemp$KpS)

}
```

```r
mean(accuracy$pos_function)
```

```
[1] 0.1594269
```

Now, let's do the same procedure for our algorithm that did not use position as a variable.

```r
accuracy <- accuracy |>
  mutate(no_position = NULL)

for(i in 1:20) {

  set.seed(i)
  training_rows <- sample(1:nrow(nescac_stats),
                          size = nrow(nescac_stats)/2)
  train_data <- nescac_stats[training_rows, ]
  test_data <- nescac_stats[-training_rows, ]

  k <- 6

  ansTemp <- data.frame(KpS = 1:nrow(test_data))

  for(j in 1:nrow(test_data)) {
  prediction <- kills_per_set(k,
                       test_data$TAperS[j],
                       test_data$PCT[j],
                       train_data)
  ansTemp$KpS[j] <- as.numeric(abs(test_data$KperS[j] - prediction))
  }

  accuracy$no_position[i] <- mean(ansTemp$KpS)

}

mean(accuracy$no_position)
```

```
[1] 0.160858
```

Since $0.1594269 > 0.160858$, we can say that yes, adding a position variable did make our algorithm slightly more accurate. However, this difference is so small that it seems position is not the most important variable we have to predict kills per set.

Conclusion

Overall, the kNN algorithm we created using attempts per set, hitting percentage, and position to predict kills per set is quite accurate. We found that six was the optimal value for k, which means that the algorithm is more complex, and finds values that are more accurate when analyzing only neighbors that are fairly close to the input. Using kNN to predict this relationship makes sense because in a real world situation, usually players with similar attempts per set, hitting percentage, and positions will have similar kills per set. The main disadvantage is that since there are so many other variables that affect a player's stats in a volleyball game, we will never be able to use all of them in one single kNN algorithm. I know volleyball and volleyball stats very well, so I could have predicted that attempts per set would give the best prediction, but before this assignment I would have definitely thought that position would have had more of an impact.

In a longer and more in-depth research assignment on this topic it would be interesting to isolate the other two variables the way we did with position, and see which is actually the most important when predicting kills per set. It could also be useful to include more data outside of just this year, or even use other conferences' or divisions' stats.

Lastly, we can look beyond just predicting kills per set for random inputs, and use context to find out some pretty interesting information using our algorithm. If we take a player and input their stats into our kNN function, we will get a prediction for their kills per set. We can use this value to analyze how well this player is doing compared to their peers (since this dataset is only other players in the conference). If their actual kills per set is higher than this prediction, then the player is one of the better performers in the NESCAC compared to other players with similar stats. It could also be interesting to take a single player's information from a previous year to see how they did compared to the competition level of this year. A similar process with a player in another conference could create an interesting analysis of competition levels across conferences.