K-Nearest Neighbors

Applied Machine Learning for Educational Data Science

true

11/01/2021

Contents

[Updated: Mon, Nov 15, 2021 - 15:16:43]

1. Distance Between Two Vectors

Measuring distance between two data points is at the core of K Nearest Neighbors (KNN) algorithm, and it is important to understand the concept of distance between two vectors.

Imagine that each observation in a dataset lives in an P-dimensional space, where P is the number of predictors.

- Obsevation 1: $\mathbf{A} = (A_1, A_2, A_3, ..., A_P)$
- Obsevation 2: $\mathbf{B} = (B_1, B_2, B_3, ..., B_P)$

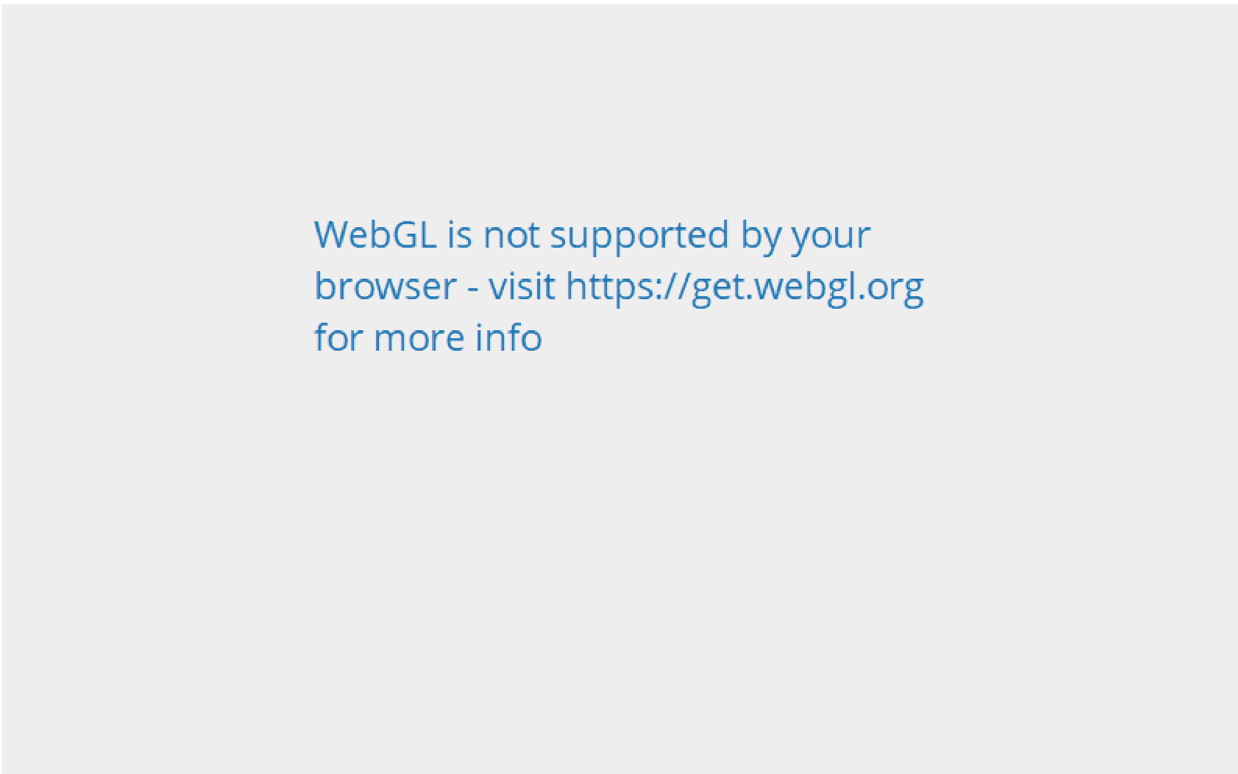
A general definition of distance between two vectors is the **Minkowski Distance**. The Minkowski Distance can be defined as

$$\left(\sum_{i=1}^{P} |A_i - B_i|^q\right)^{\frac{1}{q}},$$

where q can take any positive value.

For the sake of simplicity, suppose that we have two observations and three predictors, and we observe the following values for the two observations on these three predictors.

- Observation 1: (20,25,30)
- Observation 2: (80,90,75)

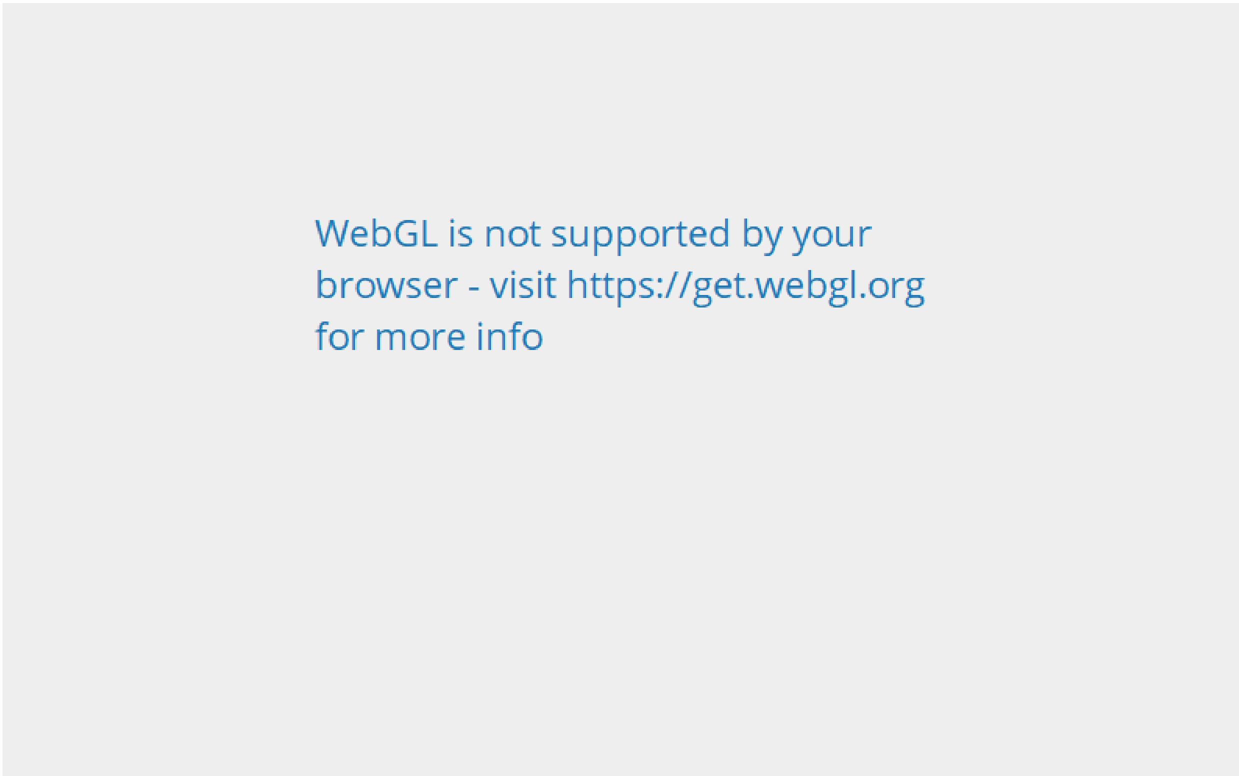


If we assume that the $q = 1$ for the Minkowski equation above, then the we can calculate the distance as the following:

```
A <- c(20,25,30)
B <- c(80,90,75)


sum(abs(A - B))
```

```
[1] 170
```

When q is equal to 1 for the Minkowski equation, it becomes a special case and is known as **Manhattan Distance**. Manhattan Distance between these two data points is visualized below.

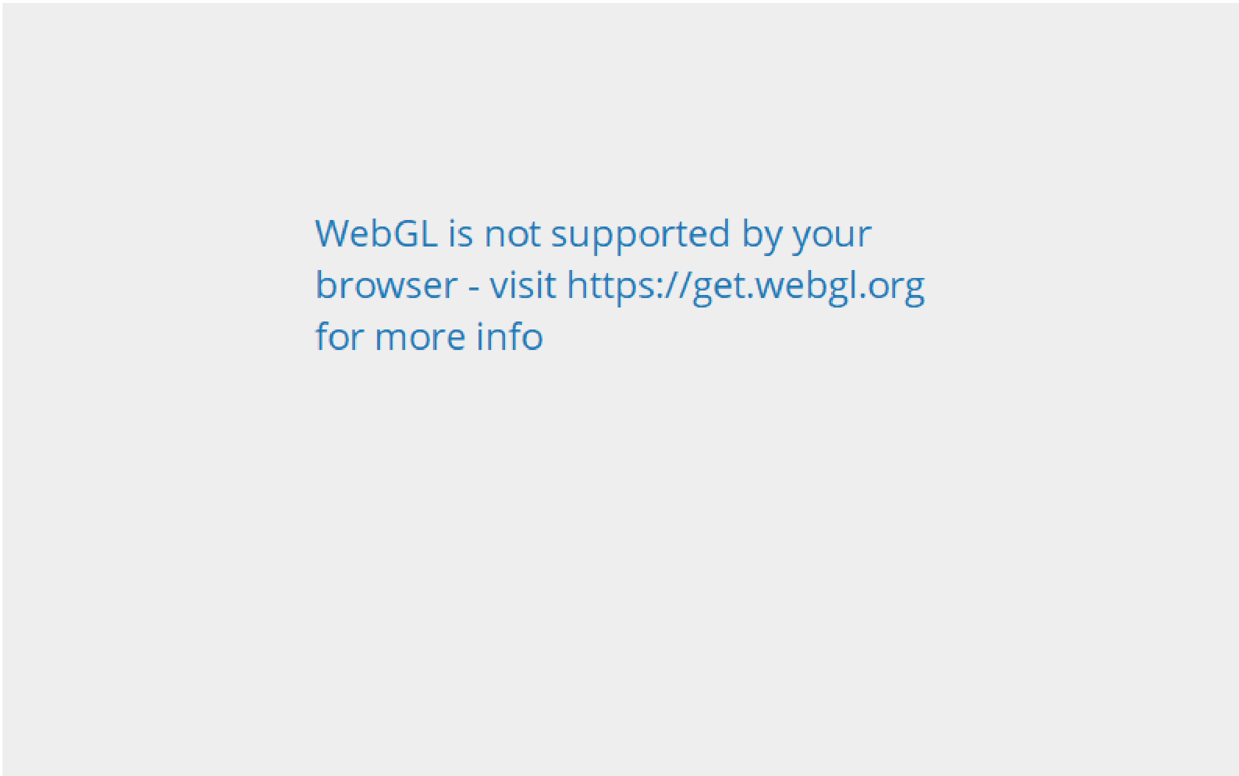


If we assume that the $q = 2$ for the Minkowski equation above, then the we can calculate the distance as the following:

```r
A <- c(20,25,30)
B <- c(80,90,75)


(sum(abs(A - B)^2))^(1/2)
```

```
[1] 99.24717
```

When q is equal to 2 for the Minkowski equation, it is also a special case and is known as **Euclidian Distance**. Euclidian Distance between these two data points is visualized below.



2. K-Nearest Neighbors

Given N observations in a dataset, a distance between any observation and $N-1$ remaining observations using Minkowski distance (with a user-defined choice of q value). Then, for any given observation, we can rank order the remaining observations based on how close they are to the given observation and then decide the K nearest neighbors ($K = 1, 2, 3, ..., N-1$), K observations closest to the given observation based on their distance.

Suppose that there are 10 observations measured on three predictor variables (X1, X2, and X3) with the following values.

```
d <- data.frame(x1 =c(20,25,30,42,10,60,65,55,80,90),
                x2 =c(10,15,12,20,45,75,70,80,85,90),
                x3 =c(25,30,35,20,40,80,85,90,92,95),
                label= c('A','B','C','D','E','F','G','H','I','J'))

d
```

```
   x1 x2 x3 label
1  20 10 25     A
2  25 15 30     B
3  30 12 35     C
4  42 20 20     D
5  10 45 40     E
6  60 75 80     F
7  65 70 85     G
8  55 80 90     H
```

```
9   80 85 92      I
10  90 90 95      J
```

WebGL is not supported by your browser - visit https://get.webgl.org for more info

Given that there 10 observations, we can calculate the distance between all 45 pairs of observations (e.g., Euclidian distance).

```r
dist <- as.data.frame(t(combn(1:10,2)))
dist$euclidian <- NA

for(i in 1:nrow(dist)){

  a <- d[dist[i,1],1:3]
  b <- d[dist[i,2],1:3]
  dist[i,]$euclidian <- sqrt(sum((a-b)^2))

}

dist
```

```
   V1 V2  euclidian
1   1  2   8.660254
2   1  3  14.282857
3   1  4  24.677925
4   1  5  39.370039
5   1  6  94.074439
6   1  7  96.046864
7   1  8 101.734950
```

```
8    1  9 117.106789
9    1 10 127.279221
10   2  3   7.681146
11   2  4  20.346990
12   2  5  35.000000
13   2  6  85.586214
14   2  7  87.464278
15   2  8  93.407708
16   2  9 108.485022
17   2 10 118.638105
18   3  4  20.808652
19   3  5  38.910153
20   3  6  83.030115
21   3  7  84.196199
22   3  8  90.961530
23   3  9 105.252078
24   3 10 115.256236
25   4  5  45.265881
26   4  6  83.360662
27   4  7  85.170417
28   4  8  93.107465
29   4  9 104.177733
30   4 10 113.265176
31   5  6  70.710678
32   5  7  75.332596
33   5  8  75.828754
34   5  9  95.937480
35   5 10 107.004673
36   6  7   8.660254
37   6  8  12.247449
38   6  9  25.377155
39   6 10  36.742346
40   7  8  15.000000
41   7  9  22.338308
42   7 10  33.541020
43   8  9  25.573424
44   8 10  36.742346
45   9 10  11.575837
```

Now, for instance, we can find the three closest observation to **Point E** (3-Nearest Neighbors). As seen below, the 3-Nearest Neighbors for **Point E** in this dataset would be **Point B**, **Point C**, and **Point A**.

```r
# Point E is the fifth observation in the dataset

loc <- which(dist[,1]==5 | dist[,2]==5)

tmp <- dist[loc,]

tmp[order(tmp$euclidian),]
```

```
   V1 V2 euclidian
12  2  5  35.00000
19  3  5  38.91015
```

```
4   1  5  39.37004
25  4  5  45.26588
31  5  6  70.71068
32  5  7  75.33260
33  5  8  75.82875
34  5  9  95.93748
35  5 10 107.00467
```

NOTE 1

The q in the Minkowski distance equation and K in the K-nearest neighbor are user-defined hyperparameters in the KNN algorithm. As a researcher and model builder, you can pick any values for q and K. They can be tuned using a similar approach applied in earlier classes for regularized regression models. One can pick a set of values for these hyperparameters and apply a grid search to find the combination that provides the best predictive performance.

It is typical to observe overfitting (high model variance, low model bias) for small values of K and underfitting (low model variance, high model bias) for large values of K. In general practice, people tend to focus their grid search for K around $\sqrt{N}$.

NOTE 2

It is important to keep in mind that the distance calculation between two observations is highly dependent on the scale of measurement for the predictor variables. If predictors are on different scales, the distance metric formula will favor the differences in predictors with larger scale. This is not an ideal situation. Therefore, it is important to center and scale all predictors prior to the KNN algorithm so each predictor contributes to the distance metric calculation in the same way.

---

3. Prediction with K-Nearest Neighbors

Given that we learn about distance calculation and how to identify K-nearest neighbors based on a distance metric, the prediction in KNN is very simple and straightforward.

Below is a list of steps for predicting an outcome for a given observation.

1. Calculate the distance between the observation and remaining $N-1$ observations in the data (with a user choice of q in Minkowski distance).
2. Rank order the observations based on the calculated distance, and choose the K-nearest neighbor. (with a user choice of K)
3. Calculate the mean of observed outcome in the smaller set of K-neareast neighbors as your prediction.

Note that, Step 3 applies regardless of the type of outcome. If the outcome variable is continous, we take the average outcome for the K-nearest neighbors as our prediction. If the outcome variable is binary variable (e.g., 0 vs. 1), then the proportion of observing each class among the K-nearest neighbors yield predicted probabilities for each class.

Below, I provide an example for both types of outcome using the Readability and Recidivism datasets.

3.1. Predicting a continuous outcome with the KNN algorithm

The code below is identical to the code we used in earlier classes for data preparation of the Readability datasets. Note that this is only to demonstrate the logic of model predictions in the context of K-nearest neighbors. So, we are using the whole dataset. In the next section, we will demonstrate the full workflow of model training and tuning with 10-fold cross-validation using the `caret::train()` function.

1. Import the data
2. Remove variables with more than 80% missingness
3. Write a recipe for processing variables
4. Apply the recipe to the dataset

```
# Import the dataset

readability <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-2021/ma

# Remove the variables with more than 80% missingness

require(finalfit)

missing_     <- ff_glimpse(readability)$Continuous
flag_na      <- which(as.numeric(missing_$missing_percent) > 80)
readability <- readability[,-flag_na]

# Write the recipe

require(recipes)

blueprint_readability <- recipe(x     = readability,
                                vars  = colnames(readability),
                                roles = c(rep('predictor',990),'outcome')) %>%
  step_zv(all_numeric()) %>%
  step_nzv(all_numeric()) %>%
  step_impute_mean(all_numeric()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric(),threshold=0.9)

# Apply the recipe

baked_read <- blueprint_readability %>%
  prep(training = readability) %>%
  bake(new_data = readability)
```

Our final dataset (`baked_read`) has 2834 observations and 888 columns (887 predictors, and the last column is target outcome). Now, suppose that we would like to predict the readability score for the first observation. The code below will calculate the Minkowski distance (with $q = 2$) between the first observation and each of the remaining 2833 observations by using the first 887 columns of the dataset (predictors).

```
dist <- data.frame(obs = 2:2834,dist = NA,target=NA)

for(i in 1:2833){

  a <- as.matrix(baked_read[1,1:887])
```

```r
  b <- as.matrix(baked_read[dist[i+1,1],1:887])
  dist[i,]$dist   <- sqrt(sum((a-b)^2))
  dist[i,]$target <- baked_read[dist[i+1,1],]$target

  #print(i)
}
```

We now rank order the observations from closest to the most distant, and then choose the 20-nearest observations (K=20). Finally, we can calculate the average of the observed outcome for the 20-nearest neighbors, this will become our prediction of the readability score for the first observation.

```r
# Rank order the observations from closest to the most distant

dist <- dist[order(dist$dist),]

# Check the 20-nearest neighbors

dist[1:20,]
```

```
      obs     dist      target
2439 2440 23.39269  0.55897492
2417 2418 25.35221 -0.21279072
2311 2312 25.90366 -0.25321371
2347 2348 26.15836 -0.15932546
2483 2484 26.28408 -0.92538206
2015 2016 26.54107  0.13989288
2318 2319 26.72020 -1.48029561
1569 1570 26.87505 -1.13367793
43     44 26.92854 -0.58635946
13     14 26.97686  0.24580571
2262 2263 27.27469 -0.90345299
117   118 27.28308 -0.20640822
2529 2530 27.31948 -0.32031046
1132 1133 27.54420  0.38387363
123   124 27.69614 -0.09360404
2152 2153 27.70797 -1.11412508
124   125 27.82294 -0.36537882
2267 2268 27.83390 -0.23113845
2520 2521 27.85822 -0.63588777
1243 1244 27.86323  0.70560033
```

```r
# Mean target for the 20-nearest observations

mean(dist[1:20,]$target)
```

```
[1] -0.3293602
```

```r
# Check the actual observed value of reability for the first observation

readability[1,]$target
```

```
[1] -0.3402591
```

3.2. Predicting a binary outcome with the KNN algorithm

We can follow the same procedures to predict the recidivism in the second year after the initial release from prison for an individual.

```
# Import data

recidivism <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-2021/main

# Write the recipe


  # List of variable types

  outcome <- c('Recidivism_Arrest_Year2')

  categorical <- c('Residence_PUMA',
                   'Prison_Offense',
                   'Age_at_Release',
                   'Supervision_Level_First',
                   'Education_Level',
                   'Prison_Years',
                   'Gender',
                   'Race',
                   'Gang_Affiliated',
                   'Prior_Arrest_Episodes_DVCharges',
                   'Prior_Arrest_Episodes_GunCharges',
                   'Prior_Conviction_Episodes_Viol',
                   'Prior_Conviction_Episodes_PPViolationCharges',
                   'Prior_Conviction_Episodes_DomesticViolenceCharges',
                   'Prior_Conviction_Episodes_GunCharges',
                   'Prior_Revocations_Parole',
                   'Prior_Revocations_Probation',
                   'Condition_MH_SA',
                   'Condition_Cog_Ed',
                   'Condition_Other',
                   'Violations_ElectronicMonitoring',
                   'Violations_Instruction',
                   'Violations_FailToReport',
                   'Violations_MoveWithoutPermission',
                   'Employment_Exempt')

  numeric   <- c('Supervision_Risk_Score_First',
                 'Dependents',
                 'Prior_Arrest_Episodes_Felony',
                 'Prior_Arrest_Episodes_Misd',
                 'Prior_Arrest_Episodes_Violent',
                 'Prior_Arrest_Episodes_Property',
                 'Prior_Arrest_Episodes_Drug',
                 'Prior_Arrest_Episodes_PPViolationCharges',
                 'Prior_Conviction_Episodes_Felony',
                 'Prior_Conviction_Episodes_Misd',
                 'Prior_Conviction_Episodes_Prop',
                 'Prior_Conviction_Episodes_Drug',
```

```r
                 'Delinquency_Reports',
                 'Program_Attendances',
                 'Program_UnexcusedAbsences',
                 'Residence_Changes',
                 'Avg_Days_per_DrugTest',
                 'Jobs_Per_Year')

props      <- c('DrugTests_THC_Positive',
                'DrugTests_Cocaine_Positive',
                'DrugTests_Meth_Positive',
                'DrugTests_Other_Positive',
                'Percent_Days_Employed')

# Convert all nominal, ordinal, and binary variables to factors
# Leave the rest as is

for(i in categorical){

  recidivism[,i] <- as.factor(recidivism[,i])

}

# For variables that represent proportions, add/substract a small number
# to 0s/1s for logit transformation

for(i in props){
  recidivism[,i] <- ifelse(recidivism[,i]==0,.0001,recidivism[,i])
  recidivism[,i] <- ifelse(recidivism[,i]==1,.9999,recidivism[,i])
}

# Blueprint for processing variables

require(recipes)

blueprint_recidivism <- recipe(x  = recidivism,
                               vars  = c(categorical,numeric,props,outcome),
                               roles = c(rep('predictor',48),'outcome')) %>%
  step_indicate_na(all_of(categorical),all_of(numeric),all_of(props)) %>%
  step_zv(all_numeric()) %>%
  step_impute_mean(all_of(numeric),all_of(props)) %>%
  step_impute_mode(all_of(categorical)) %>%
  step_logit(all_of(props)) %>%
  step_ns(all_of(numeric),all_of(props),deg_free=3) %>%
  step_normalize(paste0(numeric,'_ns_1'),
                 paste0(numeric,'_ns_2'),
                 paste0(numeric,'_ns_3'),
                 paste0(props,'_ns_1'),
                 paste0(props,'_ns_2'),
                 paste0(props,'_ns_3')) %>%
  step_dummy(all_of(categorical),one_hot=TRUE)  %>%
  step_num2factor(Recidivism_Arrest_Year2,
                transform = function(x) x + 1,
                levels=c('No','Yes'))
```

```r
# Apply the recipe

baked_recidivism <- blueprint_recidivism %>%
  prep(training = recidivism) %>%
  bake(new_data = recidivism)
```

The final dataset (`baked_recidivism`) has 18111 observations and 166 columns (165 predictors, and the first column is the outcome variable). Now, suppose that we would like to predict the probability of Recidivism for the first individual. The code below will calculate the Minkowski distance (with $q = 2$) between the first individual and each of the remaining 18,110 individuals by using values of the 166 predictors in this dataset.

```r
dist2 <- data.frame(obs = 2:18111,dist = NA,target=NA)

for(i in 1:18110){

  a <- as.matrix(baked_recidivism[1,2:165])
  b <- as.matrix(baked_recidivism[dist2[i+1,1],2:165])
  dist2[i,]$dist   <- sqrt(sum((a-b)^2))
  dist2[i,]$target <- as.character(baked_recidivism[dist2[i+1,1],]$Recidivism_Arrest_Year2)

  #print(i)
}
```

We now rank order the individuals from closest to the most distant, and then choose the 100-nearest observations (K=100). Then, we calculate proportion of individuals who were recidivated (YES) and not recidivated (NO) among these 100-nearest neighbors. These proportions are the predictions for the probability of being recidivated or not recidivated for the first individual.

```r
# Rank order the observations from closest to the most distant

dist2 <- dist2[order(dist2$dist),]

# Check the 100-nearest neighbors

dist2[1:100,]
```

```
        obs     dist target
7068   7069 7.659967     No
4525   4526 7.755362     No
8444   8445 7.859353     No
11530 11531 8.081124    Yes
9730   9731 8.090887     No
6662   6663 8.480201     No
4041   4042 8.484186     No
595     596 8.538511     No
645     646 8.558983    Yes
9111   9112 8.663452     No
1696   1697 8.687675     No
1770   1771 8.697517     No
2158   2159 8.742357     No
1027   1028 8.765372    Yes
5116   5117 8.775667     No
```

3689	3690	8.806331	No
829	830	8.821060	No
8139	8140	8.839652	Yes
384	385	8.856510	Yes
8467	8468	8.861272	No
7313	7314	8.874215	No
8766	8767	8.890606	No
4093	4094	8.927379	No
11385	11386	8.942428	Yes
2713	2714	8.949004	No
4422	4423	8.956050	Yes
1515	1516	8.963143	Yes
13840	13841	9.022972	No
1574	1575	9.051930	No
14383	14384	9.055670	No
8186	8187	9.067094	No
13020	13021	9.068971	No
9070	9071	9.073520	No
6355	6356	9.097278	No
3966	3967	9.108941	Yes
13820	13821	9.109367	No
2947	2948	9.109936	No
894	895	9.116890	No
11091	11092	9.117779	No
14202	14203	9.122793	No
6022	6023	9.135977	No
3434	3435	9.141237	Yes
11015	11016	9.153111	Yes
9946	9947	9.166294	No
12473	12474	9.174083	No
563	564	9.179573	Yes
1572	1573	9.187760	No
7785	7786	9.188086	No
807	808	9.199478	Yes
14783	14784	9.205327	Yes
15152	15153	9.205441	No
10816	10817	9.206677	Yes
3029	3030	9.221680	No
986	987	9.224630	No
2946	2947	9.231919	No
14505	14506	9.241374	No
13910	13911	9.250168	No
5306	5307	9.254097	Yes
4870	4871	9.278566	Yes
14526	14527	9.278633	Yes
4794	4795	9.282708	No
1561	1562	9.287491	No
9025	9026	9.287869	Yes
10167	10168	9.289060	No
2	3	9.305465	No
288	289	9.307759	Yes
599	600	9.336480	No
4778	4779	9.353857	No
4166	4167	9.367567	No

```
7297   7298 9.371710    No
4644   4645 9.374108    No
2299   2300 9.375541    No
1447   1448 9.386387    No
9676   9677 9.399002   Yes
9134   9135 9.412505    No
4583   4584 9.420181   Yes
7958   7959 9.425067    No
10761 10762 9.425995   Yes
7201   7202 9.437474    No
5314   5315 9.445364    No
1917   1918 9.448661   Yes
15125 15126 9.455830   Yes
3170   3171 9.460053    No
1388   1389 9.461795   Yes
9286   9287 9.465960    No
615     616 9.468258    No
548     549 9.469860    No
4229   4230 9.475141    No
4743   4744 9.476835    No
3641   3642 9.481367    No
4874   4875 9.481934   Yes
9727   9728 9.485522    No
15326 15327 9.486961    No
9381   9382 9.489079   Yes
3362   3363 9.489238   Yes
8641   8642 9.490666   Yes
2745   2746 9.491828    No
5741   5742 9.493817    No
1578   1579 9.496209    No
8963   8964 9.515531    No
```

```
# Mean target for the 100-nearest observations

table(dist2[1:100,]$target)
```

```
 No Yes 
 70  30 
```

```
  # This indicates that the predicted probability of being recidivated is 0.30
  # for the first individual given the observed data for 100 most similar
  # observations
```

```
# Check the actual observed outcome for the first individual

recidivism[1,]$Recidivism_Arrest_Year2
```

```
[1] 0
```

4. Kernels to Weight the Neighbors

In the previous section, we tried to understand how KNN predicts a target outcome by simply averaing the observed value for the target outcome from K-nearest neighbors. This was a simple average by equally weighting each neighbor.

Another way of averaging the target outcome from K-nearest neighbors would be to weight each neighbor according to its distance, and calculate a weighted average. A simple way to weight each neighbor is to use the inverse of distance. For instance, consider the earlier example where we find the 20-nearest neighbor for the first observation in the reeadability dataset.

```
dist <- dist[order(dist$dist),]

k_neighbors <- dist[1:20,]

k_neighbors
```

```
      obs     dist      target
2439 2440 23.39269  0.55897492
2417 2418 25.35221 -0.21279072
2311 2312 25.90366 -0.25321371
2347 2348 26.15836 -0.15932546
2483 2484 26.28408 -0.92538206
2015 2016 26.54107  0.13989288
2318 2319 26.72020 -1.48029561
1569 1570 26.87505 -1.13367793
43     44 26.92854 -0.58635946
13     14 26.97686  0.24580571
2262 2263 27.27469 -0.90345299
117   118 27.28308 -0.20640822
2529 2530 27.31948 -0.32031046
1132 1133 27.54420  0.38387363
123   124 27.69614 -0.09360404
2152 2153 27.70797 -1.11412508
124   125 27.82294 -0.36537882
2267 2268 27.83390 -0.23113845
2520 2521 27.85822 -0.63588777
1243 1244 27.86323  0.70560033
```

We can assign a weight to each neighbor by taking the inverse of their distance and rescaling them such that the sum of the weights are equal to 1.

```
k_neighbors$weight <- 1/k_neighbors$dist
k_neighbors$weight <- k_neighbors$weight/sum(k_neighbors$weight)


k_neighbors
```

```
      obs     dist      target     weight
2439 2440 23.39269  0.55897492 0.05732923
2417 2418 25.35221 -0.21279072 0.05289814
2311 2312 25.90366 -0.25321371 0.05177203
2347 2348 26.15836 -0.15932546 0.05126793
2483 2484 26.28408 -0.92538206 0.05102271
2015 2016 26.54107  0.13989288 0.05052867
2318 2319 26.72020 -1.48029561 0.05018993
1569 1570 26.87505 -1.13367793 0.04990073
```

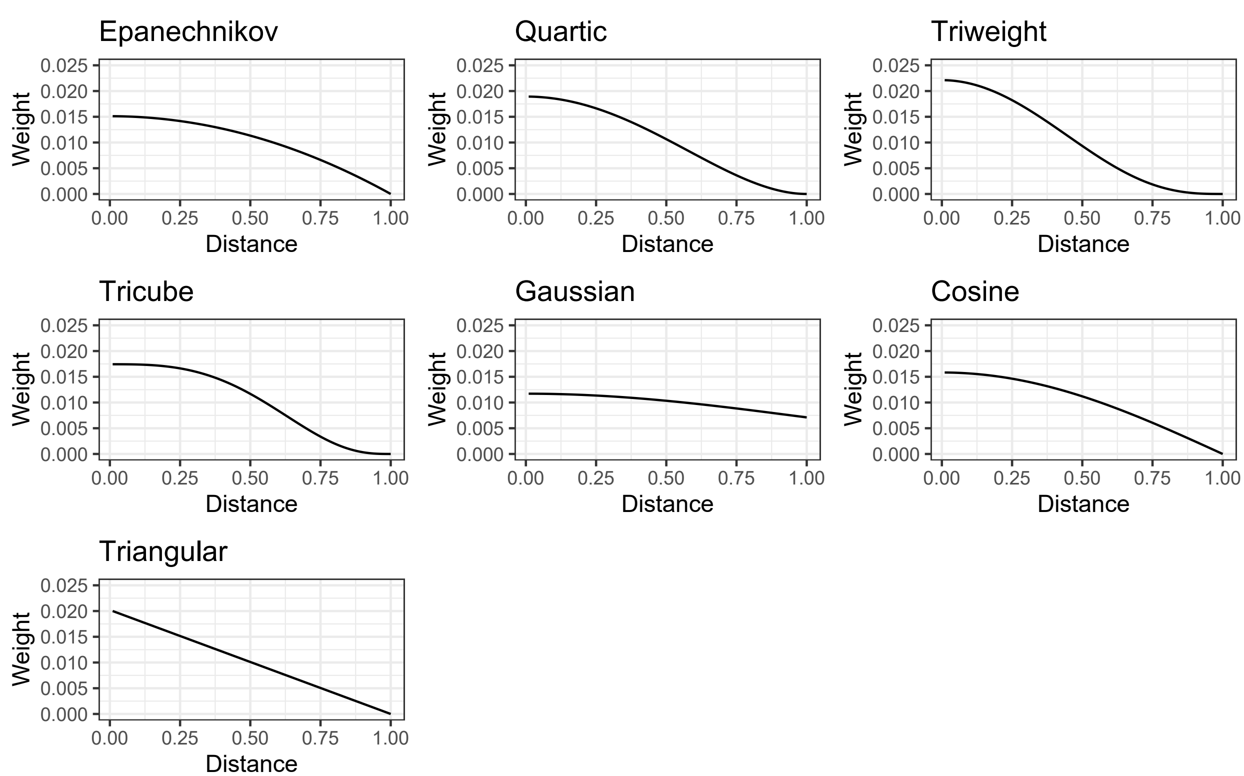
```
43      44 26.92854 -0.58635946 0.04980163
13      14 26.97686  0.24580571 0.04971241
2262 2263 27.27469 -0.90345299 0.04916958
117    118 27.28308 -0.20640822 0.04915446
2529 2530 27.31948 -0.32031046 0.04908897
1132 1133 27.54420  0.38387363 0.04868847
123    124 27.69614 -0.09360404 0.04842137
2152 2153 27.70797 -1.11412508 0.04840069
124    125 27.82294 -0.36537882 0.04820069
2267 2268 27.83390 -0.23113845 0.04818172
2520 2521 27.85822 -0.63588777 0.04813964
1243 1244 27.86323  0.70560033 0.04813100
```

Then, we can compute a weighted average of the target scores instead of a simple average.

```r
# Weighted Mean target for the 20-nearest observations

sum(k_neighbors$target*k_neighbors$weight)
```

```
[1] -0.3239415
```

There are a number of different kernel function to assign weight to K-nearest neighbors (e.g., epanechnikov, quartic, triweight, tricube, gaussian, cosine). For all of them, closest neighbors are assigned higher weights while the weight gets smaller as the distance increases. They slightly differ the way they assign the weight. Below is a demonstration how assigned weight changes as a function of distance for different kernel functions.



NOTE 3

Which kernel function should we use for weighting the distance? The type of kernel function can also be considered as a hyperparameter to tune.

---

5. Parallel Processing with the `caret::train()` function

The KNN algorithm can become computationally quite intensive and grid search with so many combinations may take days particularly when you have a very large sample size. One way to reduce the computational time and make it reasonable is to use parallel processing when you have access to multiple core computers (or computing clusters).

The `caret` package is designed in a way to take advantage of multiple cores in a computer via another package `doParallel`. If you are worried about the computation time for fitting any model using the `caret::train` function, you can run the following lines of code to use as many cores as available in your computing device.

```
require(doParallel)

ncores <- 15    # depends on the number of cores available in your computer

cl <- makePSOCKcluster(ncores)

registerDoParallel(cl)
```

Once you run these lines of code, anytime you run the `caret::train` function to train any model, it will use all registered cores to do the computations. This could significantly reduce the computational time from days to hours or from hours to minutes.

After you are done with training the model, you can run the following line of code to stop the computational cluster registered for the use of your current R session.

```
stopCluster(cl)
```

You can find more information about parallel processing with the caret package at this link.

6. K-Nearest Neighbors Algorithm to Predict Readability Scores

In this section, we implement the KNN algorithm to build a prediction model for the readability score. The first part of the code is identical to the code we use in earlier classes for the following steps:

1. Split the data into train and test sets
2. Create the list of row indices for 10-fold cross validation
3. Set the object for cross validation settings.

```
# Train and Test Split

  set.seed(10152021)  # for reproducibility

  loc      <- sample(1:nrow(readability), round(nrow(readability) * 0.9))
```

```r
  read_tr  <- readability[loc, ]
  read_te  <- readability[-loc, ]

# Create the row indices for 10-folds

    # Randomly shuffle the training data

      read_tr = read_tr[sample(nrow(read_tr)),]

    # Create 10 folds with equal size

      folds = cut(seq(1,nrow(read_tr)),breaks=10,labels=FALSE)

    # Create the list for each fold

      my.indices <- vector('list',10)
      for(i in 1:10){
        my.indices[[i]] <- which(folds!=i)
      }

# Cross-validation settings

  cv <- trainControl(method = "cv",
                     index  = my.indices)
```

We will use the **kknn** package to implement the KNN algorithm, and this method is available through the `caret::train()` function. First, let's check which hyperparameters are available to tune for this method.

```r
# install.package('kknn')

require(caret)
require(kknn)


getModelInfo()$kknn$parameters
```

```
  parameter     class            label
1      kmax   numeric Max. #Neighbors
2  distance   numeric         Distance
3    kernel character           Kernel
```

There are three hyperparameters available to optimize for the `kknn` method. These are the same parameters discussed earlier:

- `kmax`: the choice of K for the K-nearest neighbors (numeric).
- `distance`: the choice of power value (q) for the Minkowski distance (numeric)
- `kernel`: the choice of kernel function for weigted predictions (character)

The possible kernel functions available are "rectangular","triangular", "epanechnikov", "biweight", "triweight", "cos", "inv", and "gaussian". The "rectangular" indicates a simple average with no weight applied (equally weighted).

The next step is to create a matrix for hyperparameter grid search. Let's consider the following values for these three hyperparameters:

- `kmax`: 2, 3, 4,. . . , 25
- `distance`: 1, 2, 3
- `kernel`: rectangular, epanechnikov

```
# Hyperparameter Tuning Grid

  grid <- expand.grid(kmax     = 3:25,
                      distance = c(1,2,3),
                      kernel   = c('epanechnikov','rectangular'))
  grid
```

```
    kmax distance       kernel
1      3        1 epanechnikov
2      4        1 epanechnikov
3      5        1 epanechnikov
4      6        1 epanechnikov
5      7        1 epanechnikov
6      8        1 epanechnikov
7      9        1 epanechnikov
8     10        1 epanechnikov
9     11        1 epanechnikov
10    12        1 epanechnikov
11    13        1 epanechnikov
12    14        1 epanechnikov
13    15        1 epanechnikov
14    16        1 epanechnikov
15    17        1 epanechnikov
16    18        1 epanechnikov
17    19        1 epanechnikov
18    20        1 epanechnikov
19    21        1 epanechnikov
20    22        1 epanechnikov
21    23        1 epanechnikov
22    24        1 epanechnikov
23    25        1 epanechnikov
24     3        2 epanechnikov
25     4        2 epanechnikov
26     5        2 epanechnikov
27     6        2 epanechnikov
28     7        2 epanechnikov
29     8        2 epanechnikov
30     9        2 epanechnikov
31    10        2 epanechnikov
32    11        2 epanechnikov
33    12        2 epanechnikov
34    13        2 epanechnikov
35    14        2 epanechnikov
36    15        2 epanechnikov
37    16        2 epanechnikov
38    17        2 epanechnikov
39    18        2 epanechnikov
40    19        2 epanechnikov
41    20        2 epanechnikov
42    21        2 epanechnikov
```

43	22	2	epanechnikov
44	23	2	epanechnikov
45	24	2	epanechnikov
46	25	2	epanechnikov
47	3	3	epanechnikov
48	4	3	epanechnikov
49	5	3	epanechnikov
50	6	3	epanechnikov
51	7	3	epanechnikov
52	8	3	epanechnikov
53	9	3	epanechnikov
54	10	3	epanechnikov
55	11	3	epanechnikov
56	12	3	epanechnikov
57	13	3	epanechnikov
58	14	3	epanechnikov
59	15	3	epanechnikov
60	16	3	epanechnikov
61	17	3	epanechnikov
62	18	3	epanechnikov
63	19	3	epanechnikov
64	20	3	epanechnikov
65	21	3	epanechnikov
66	22	3	epanechnikov
67	23	3	epanechnikov
68	24	3	epanechnikov
69	25	3	epanechnikov
70	3	1	rectangular
71	4	1	rectangular
72	5	1	rectangular
73	6	1	rectangular
74	7	1	rectangular
75	8	1	rectangular
76	9	1	rectangular
77	10	1	rectangular
78	11	1	rectangular
79	12	1	rectangular
80	13	1	rectangular
81	14	1	rectangular
82	15	1	rectangular
83	16	1	rectangular
84	17	1	rectangular
85	18	1	rectangular
86	19	1	rectangular
87	20	1	rectangular
88	21	1	rectangular
89	22	1	rectangular
90	23	1	rectangular
91	24	1	rectangular
92	25	1	rectangular
93	3	2	rectangular
94	4	2	rectangular
95	5	2	rectangular
96	6	2	rectangular

```
97      7         2  rectangular
98      8         2  rectangular
99      9         2  rectangular
100    10         2  rectangular
101    11         2  rectangular
102    12         2  rectangular
103    13         2  rectangular
104    14         2  rectangular
105    15         2  rectangular
106    16         2  rectangular
107    17         2  rectangular
108    18         2  rectangular
109    19         2  rectangular
110    20         2  rectangular
111    21         2  rectangular
112    22         2  rectangular
113    23         2  rectangular
114    24         2  rectangular
115    25         2  rectangular
116     3         3  rectangular
117     4         3  rectangular
118     5         3  rectangular
119     6         3  rectangular
120     7         3  rectangular
121     8         3  rectangular
122     9         3  rectangular
123    10         3  rectangular
124    11         3  rectangular
125    12         3  rectangular
126    13         3  rectangular
127    14         3  rectangular
128    15         3  rectangular
129    16         3  rectangular
130    17         3  rectangular
131    18         3  rectangular
132    19         3  rectangular
133    20         3  rectangular
134    21         3  rectangular
135    22         3  rectangular
136    23         3  rectangular
137    24         3  rectangular
138    25         3  rectangular
```

There are a total of 138 combinations we would like to search. Note that, this means a total of 1380 model fitted with 10-fold cross validation for each combination. Also, the KNN algorithm is very slow to fit (because you have to calculate N*(N-1)/2 distances). This may take very long. Therefore, I will also use parallel processing with 15 cores.

```
require(doParallel)

ncores <- 15    # depends on the number of cores available in your computer

cl <- makePSOCKcluster(ncores)
```

```r
registerDoParallel(cl)
```

Now, we can fit the model with these settings.

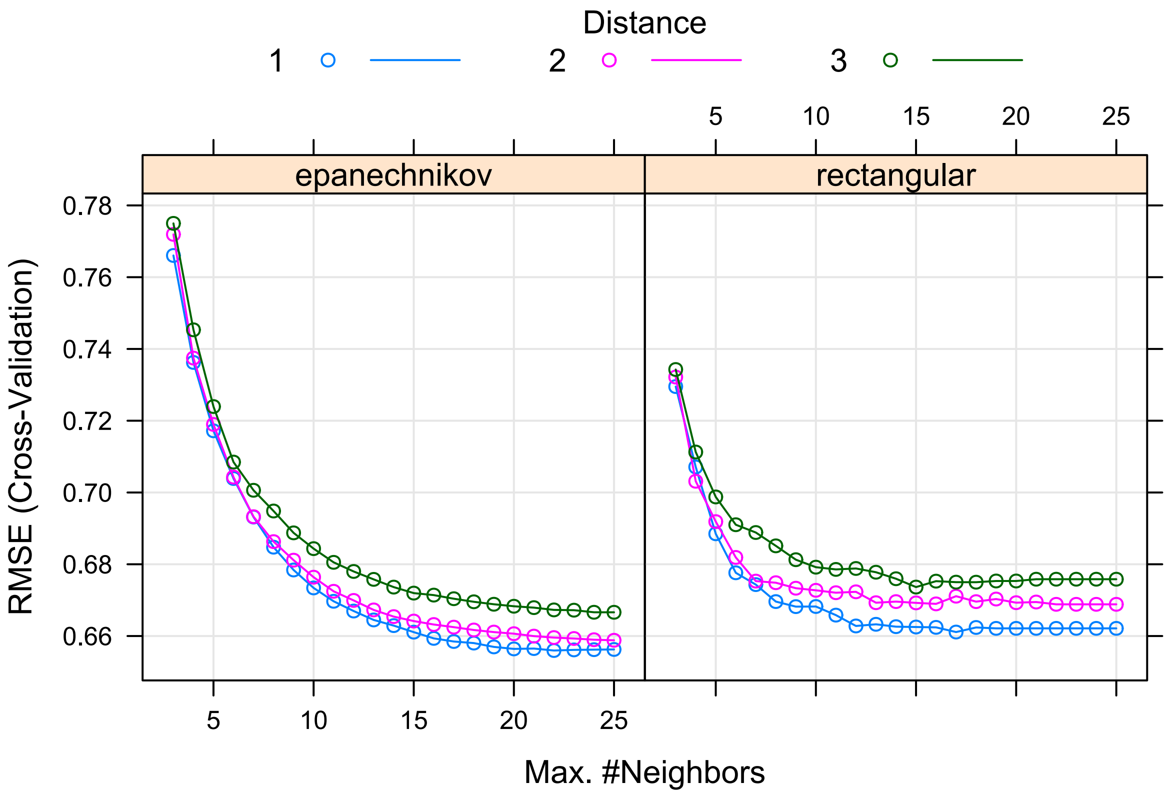
```r
# Train the model

  caret_knn_readability <- caret::train(blueprint_readability,
                                        data      = read_tr,
                                        method    = "kknn",
                                        trControl = cv,
                                        tuneGrid  = grid)

  caret_knn_readability$times
```

```
$everything
    user   system  elapsed 
  208.17     5.86 11534.15 

$final
   user  system elapsed 
 186.16    0.50  186.25 

$prediction
[1] NA NA NA
```

The training took 11534.15 seconds (~ 3 hours 12 minutes) with 15 cores running at the same time (!).

Let's check the best combination of hyperparameters that optimizes the 10-fold cross validated performance metric (RMSE).

```r
plot(caret_knn_readability)
```




```
caret_knn_readability$bestTune
```

```
    kmax distance       kernel
115   22        1 epanechnikov
```

Now, we can check the performance of the KNN algorithm on the test dataset.

```
predicted_te <- predict(caret_knn_readability,read_te)

# R-square

cor(read_te$target,predicted_te)^2
```

```
[1] 0.6226773
```

```
# RMSE

sqrt(mean((read_te$target - predicted_te)^2))
```

```
[1] 0.6297403
```

```r
# MAE

mean(abs(read_te$target - predicted_te))
```

```
[1] 0.4998969
```

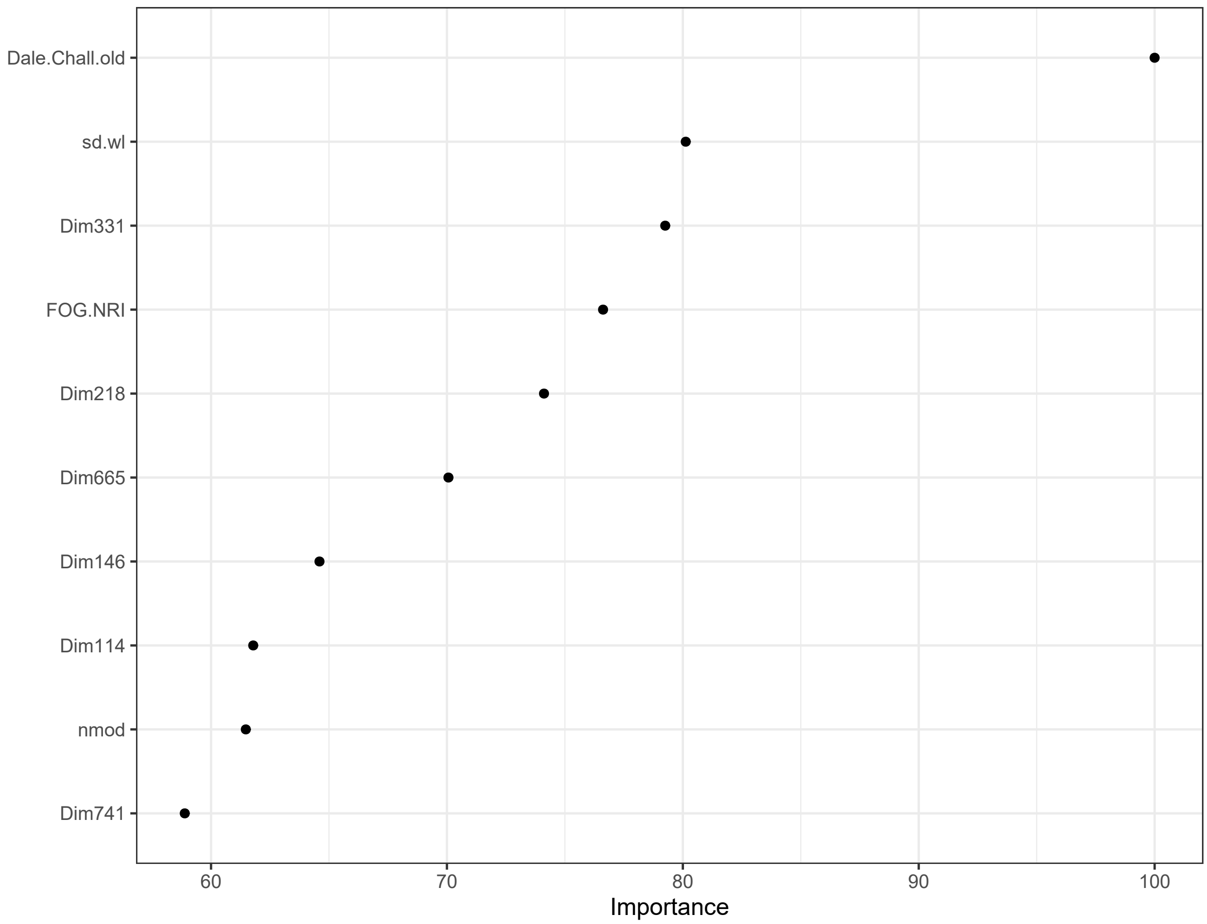
The table below provides a comparison to the models we discussed earlier.

	R-square	MAE	RMSE
Linear Regression	0.644	0.522	0.644
Ridge Regression	0.727	0.435	0.536
Lasso Regression	0.725	0.434	0.538
KNN	0.623	0.500	0.629

Variable Importance

You can ask for the variable importance plot from the KNN algorithmsimilar to other models (although I am not sure how they calculate the variable importance in the context of KNN).

```r
require(vip)

vip(caret_knn_readability,
    num_features = 10,
    geom = "point") +
  theme_bw()
```


Dale.Chall.old
sd.wl
Dim331
FOG.NRI
Dim218
Dim665
Dim146
Dim114
nmod
Dim741
60
70
80
90
100
Importance