Ejemplo KNN

Opciones de código

Aida Calviño - Junio 2025

Tareas iniciales: librerías y datos

Comenzamos cargando las librerías que vamos a necesitar:

▼ Mostrar el código

```
library(caret)
library(pROC)
library(kknn)
library(iml)
# Dado que la validación cruzada puede aprovechar los beneficios de la
# paralelización, lo ponemos (si da problemas en tu ordenador, puedes obviarlo)
library(doParallel)
cluster <- makeCluster(detectCores() - 1)
registerDoParallel(cluster)
```

En este documento vamos a ver cómo construir modelos KNN (k vecinos más próximos), un modelo sin asunciones teóricas, no paramétrico y adecuado para relaciones no lineales entre variables *input* y objetivo.

Vamos a contar con dos variables dependientes diferentes, que nos van a permitir ilustrar cómo funcionan estos modelos cuando se aplican sobre variables dependientes cuantitativas y cualitativas.

Vamos a continuar utilizando el conjunto de datos del tema anterior, cuyas variables independiente se resumen en la siguiente tabla:

Variable	Descripción	Codificación
Genero		
Edad	En años	
Altura	En metros	
Hist_fam	¿Algún miembro de su familia ha padecido sobrepeso?	
ComidaCalorica	¿Consume frecuentemente comida altamente calórica?	
Verduras	¿Consume verdura en todas las comidas?	1 = Nunca; 2 = A veces; 3 = Siempre
NumeroComidas	¿Cuántas comidas realiza al día?	1 = 1; 2 = 2; 3 = 3; 4 = 4 ó más
EntreHoras	¿Suele comer "entre horas"?	
Agua	¿Cuánta agua consume diariamente?	1 = Menos de 1litro; 2 = Entre 1 y 2 litros; 3 = Más de 2 litros
ActividadFisica	¿Cuántas veces a la semana practica alguna actividad deportiva?	
Alcohol	¿Consume alcohol frecuentemente?	

Para el caso de variable dependiente cualitativa, vamos a modelizar *Sobrepeso*; variable que recoge si el individuo sufre sobrepeso (teniendo en cuenta el IMC). Esta variable está codificada como “1” (sí se tiene

sobrepeso) y “0” (no se tiene sobrepeso).

En cuanto a la variable cuantitativa, pretendemos modelizar el Índice de Masa Corporal (IMC) de los encuestados.

Como en ocasiones anteriores, una vez cargados los datos debemos hacer un *summary* para ver si los datos tienen errores y/o datos ausentes (pero no es así en este caso):

▼ Mostrar el código

```
datosClasif<-readRDS("DatosEjemploBin")
summary(datosClasif)
```

```
   Genero          Edad           Altura      Sobrepeso Hist_fam
Female:1029   Min.   :14.00   Min.   :1.450   0: 560    no : 381
Male  :1049   1st Qu.:20.00   1st Qu.:1.629   1:1518    yes:1697
              Median :23.00   Median :1.700
              Mean   :24.33   Mean   :1.701
              3rd Qu.:26.00   3rd Qu.:1.768
              Max.   :61.00   Max.   :1.980
ComidaCalorica Verduras NumeroComidas EntreHoras Agua     Alcohol
no : 239        1:982    1:491         No : 291   1: 506   no : 633
yes:1839        2:994    2:516         Yes:1787   2:1095   yes:1445
                3:102    3:509                    3: 477
                         4:562


  ActFisica
Min.   :0.00
1st Qu.:0.00
Median :1.00
Mean   :1.62
3rd Qu.:3.00
Max.   :7.00
```

▼ Mostrar el código

```
datosRegr<-readRDS("DatosEjemploNum")
summary(datosRegr)
```

```
   Genero          Edad           Altura           IMC         Hist_fam
Female:1029   Min.   :14.00   Min.   :1.450   Min.   :13.00   no : 381
Male  :1049   1st Qu.:20.00   1st Qu.:1.629   1st Qu.:24.30   yes:1697
              Median :23.00   Median :1.700   Median :28.70
              Mean   :24.33   Mean   :1.701   Mean   :29.69
              3rd Qu.:26.00   3rd Qu.:1.768   3rd Qu.:36.00
              Max.   :61.00   Max.   :1.980   Max.   :50.80
ComidaCalorica Verduras NumeroComidas EntreHoras Agua     Alcohol
no : 239        1:982    1:491         No : 291   1: 506   no : 633
```

```
yes:1839       2:994    2:516        Yes:1787   2:1095   yes:1445
               3:102    3:509                   3: 477
                        4:562
```

```
  ActFisica
Min.   :0.00
1st Qu.:0.00
Median :1.00
Mean   :1.62
3rd Qu.:3.00
Max.   :7.00
```

Modelo del vecino más cercano (KNN)

Como ya hemos visto, esta técnica permite predecir el valor de una observación a través de la información que facilitan las observaciones más similares.

Los modelos KNN se construyen en *R* con múltiples funciones pero, debido a sus características, optamos por la función *kknn* de la librería homónima. No existen grandes diferencias en la aplicación sobre variables categóricas o cuantitativas

KNN para regresión

Antes de comenzar con el proceso de modelización, procedemos a llevar a cabo la partición entrenamiento-prueba:

▼ Mostrar el código

```
set.seed(12345)
trainIndex <- createDataPartition(datosRegr$IMC, p=0.8, list=FALSE)
data_rg_train <- datosRegr[trainIndex,]
data_rg_test <- datosRegr[-trainIndex,]
```

Empezamos mostrando cómo obtener modelos KNN para regresión, es decir, para variables dependientes cuantitativas. Para ello, intentamos predecir el IMC de los individuos. Como ya se comentó en la teoría, el principal parámetro a fijar es el número de vecinos que se va a considerar. La versión más clásica de los modelos KNN pondera por igual la información de todos los vecinos (es decir se realiza una media aritmética), pero esto se podría modificar para darle más peso a las observaciones más próximas. La función *kknn()* que vamos a utilizar permite esta opción mediante el parámetro *kernel* pero vamos a ignorarlo en este curso para simplificar. Se puede consultar la ayuda de la función si se desea información adicional.

Vamos a construir un primer modelo con 5 vecinos y después intentaremos mejorar los resultados modificando los parámetros. Cabe destacar que la función que vamos a utilizar estandariza internamente las variables y convierte en *dummy* las cualitativas, por lo que no es necesario ningún preprocesamiento:

▼ Mostrar el código

```
modelo1 <- kknn(IMC~., data_rg_train, data_rg_test, distance = 2, k=5, kernel = "rectangular")
head(modelo1$D)
```

```
          [,1]      [,2]      [,3]      [,4]     [,5]
[1,] 2.0248115 2.1862693 2.2506720 2.2571519 2.273006
[2,] 0.6828212 1.2509121 1.3264509 1.3985484 1.618601
[3,] 0.7084872 2.0110030 2.0529938 2.2432447 2.520969
[4,] 0.0000000 1.7152423 2.6075283 2.6867450 2.907287
[5,] 0.1097209 0.6066843 0.6091105 0.6334305 1.640904
[6,] 1.4814754 2.6764033 2.8368255 2.8446331 2.975654
```

▼ Mostrar el código

```
head(modelo1$CL)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 17.8 18.5 18.3 33.8 17.9
[2,] 31.3 26.4 19.4 32.3 30.9
[3,] 22.6 49.5 25.4 19.0 22.5
[4,] 21.0 25.9 22.1 21.8 21.0
[5,] 18.0 17.6 17.7 17.9 17.8
[6,] 17.9 17.5 24.4 21.6 22.8
```

▼ Mostrar el código

```
head(modelo1$fitted.values)
```

```
[1] 21.26 28.06 27.80 22.36 17.80 20.84
```

Esta función requiere que le indiquemos la fórmula (es decir, variable objetivo y las input), además de los conjuntos de datos de entrenamiento y prueba, el número de vecinos, el tipo de ponderación (rectangular implica mismo peso para todos los vecinos) y el tipo de distancia. De manera genérica, se considera la distancia Minkowski, que tiene como caso particular la distancia euclídea cuando *distance=2*. Así mismo, si *distance=1*, se estaría utilizando la distancia de *Manhattan*, que es preferible cuando existen muchas variables input y/o existen datos atípicos.

Para entender mejor el funcionamiento de este método, hemos pedido que nos muestre parte de las matrices D y CL, que contienen las distancias a los $k = 5$ vecinos más próximos de las observaciones en el conjunto de datos de prueba, y los valores de la variable objetivo, respectivamente. Cabe recordar que esos vecinos más próximos se seleccionan de entre las observaciones del conjunto de datos de entrenamiento. Finalmente, hemos pedido los valores predichos; podemos verificar que resultan de hacer la media aritmética de las filas de la matriz CL. Es interesante verificar que no todas las observaciones contienen vecinos igual de próximos.

A continuación, podemos calcular la calidad del modelo construido en prueba:

▼ Mostrar el código

```
R2(modelo1$fitted.values,data_rg_test$IMC)
```

```
[1] 0.7306233
```

Desgraciadamente, no es posible obtener una estimación realista de la calidad en entrenamiento puesto que la propia "bolsa" donde se buscan los vecinos contiene a la observación y eso hace que el resultado sea optimista por naturaleza (por ejemplo, si se pone *k=1* el modelo sería perfecto).

Para poder determinar el valor óptimo de k y si es mejor la distancia euclídea o la de Manhattan, podemos recurrir a la validación cruzada (no la hacemos repetida pues se trata de un proceso costoso computacionalmente hablando), a través de la funcion *train()* de la librería caret. En esta ocasión, en lugar de indicar directamente la fórmula a aplicar, separamos la variables objetivo (*y*) y la matriz que contiene la información de las variables input (*x*):

▼ Mostrar el código

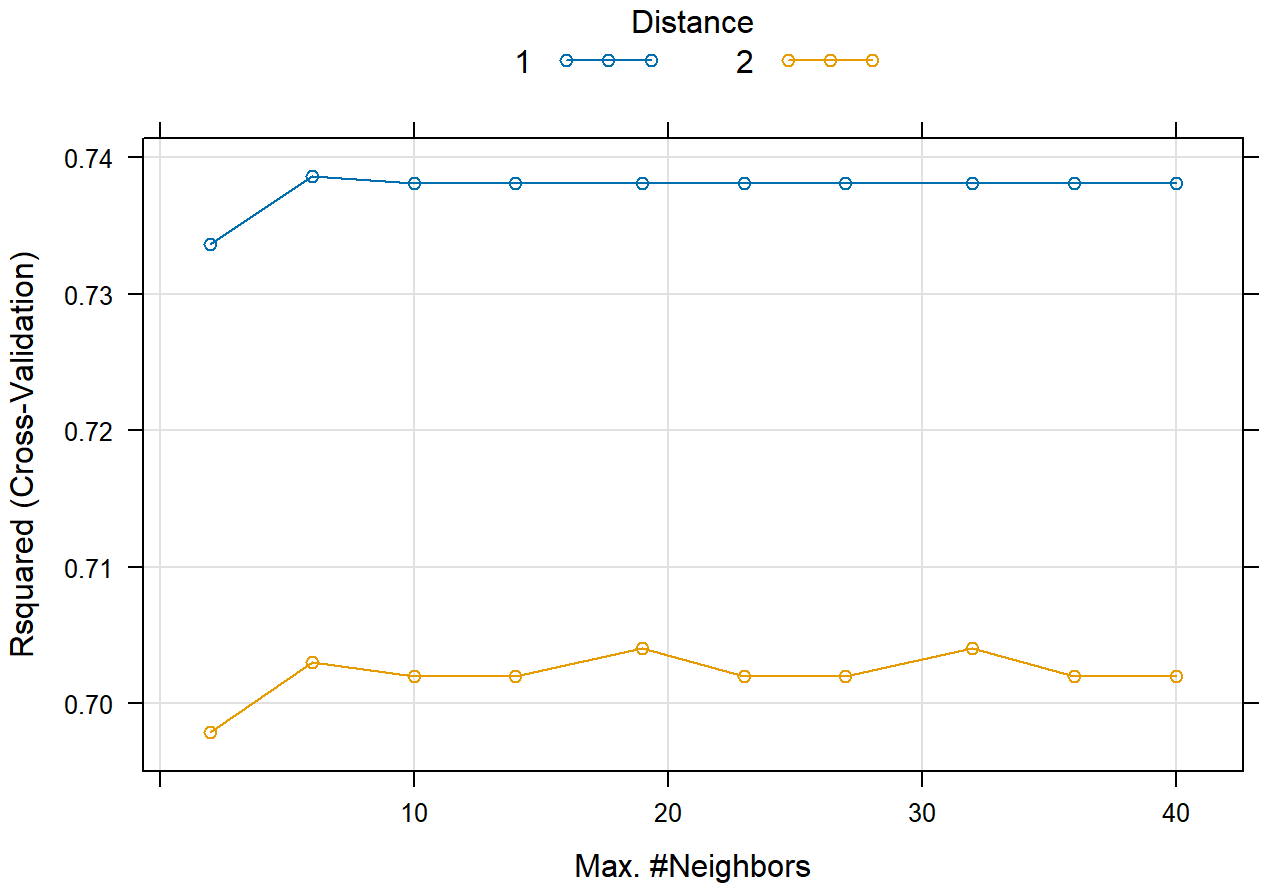
```
set.seed(12345)
# En la posición 4 está la variable objetivo
knn_tuneTodo <- train(y=data_rg_train$IMC, x = data_rg_train[,-4],
                  method = "kknn",
                  trControl = trainControl(method="cv", number = 5),
                  metric="Rsquared",
                  tuneGrid = expand.grid(kmax=floor(seq.int(2,sqrt(nrow(data_rg_train)),length.ou
knn_tuneTodo
```

```
k-Nearest Neighbors

1665 samples
  11 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 1332, 1332, 1332, 1333, 1331
Resampling results across tuning parameters:

  kmax  distance  RMSE      Rsquared   MAE
   2    1         4.224576  0.7336588  2.662974
   2    2         4.507302  0.6978697  2.939756
   6    1         4.147860  0.7385842  2.796971
   6    2         4.407893  0.7030160  3.057319
  10    1         4.149451  0.7381368  2.817355
  10    2         4.406361  0.7019949  3.094314
  14    1         4.149451  0.7381368  2.817355
  14    2         4.406361  0.7019949  3.094314
  19    1         4.149451  0.7381368  2.817355
  19    2         4.391099  0.7040076  3.085830
  23    1         4.149451  0.7381368  2.817355
  23    2         4.406361  0.7019949  3.094314
  27    1         4.149451  0.7381368  2.817355
```

```
27    2          4.406361  0.7019949  3.094314
32    1          4.149451  0.7381368  2.817355
32    2          4.391099  0.7040076  3.085830
36    1          4.149451  0.7381368  2.817355
36    2          4.406361  0.7019949  3.094314
40    1          4.149451  0.7381368  2.817355
40    2          4.406361  0.7019949  3.094314

Tuning parameter 'kernel' was held constant at a value of rectangular
Rsquared was used to select the optimal model using the largest value.
The final values used for the model were kmax = 6, distance = 1 and kernel
 = rectangular.
```

▼ Mostrar el código

```
plot(knn_tuneTodo,metric=c("Rsquared"))
```



Algunos autores recomiendan como k genérico utilizar la raíz cuadrada del número de observaciones en entrenamiento. Dado que esto puede resultar excesivo, hemos evaluado una parrilla de valores, que van desde 2 hasta esa cantidad. Vemos que, a partir de $k = 6$ no se observan diferencias en el R^2, por lo que no es necesario un número de vecinos tan elevado para estos datos. Es importante destacar que, en el caso de que el mejor valor se observe en alguno de los extremos, sería recomendable aumentar por ese lado la rejilla de valores.

En cuanto al tipo de distancia, parece preferible recurrir a la distancia de Manhattan.

Hemos llevado una búsqueda de parámetros inicial para encontrar la zona más adecuada de k donde buscar. No obstante, como ya se mencionó en la teoría, los modelos KNN están fuertemente afectados por variables input sin poder predictivo, por lo que a continuación estudiamos cómo seleccionar variables.

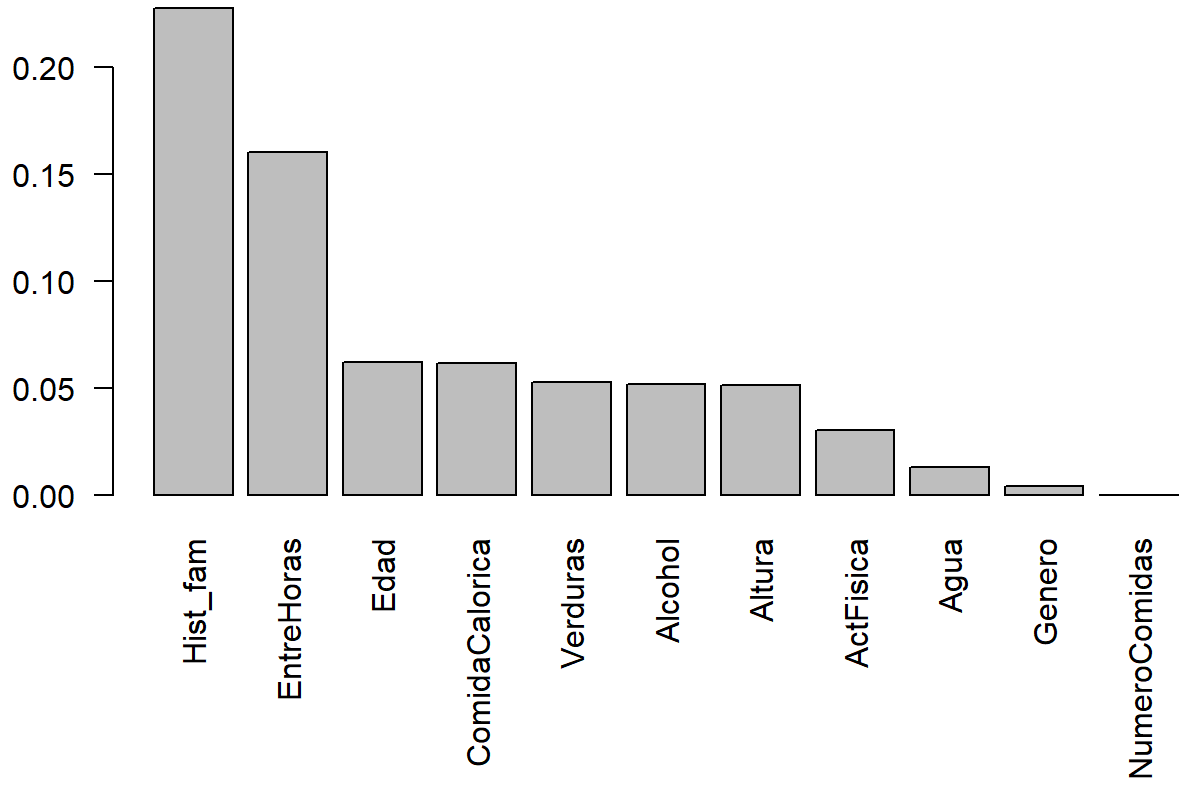
Selección de variables

El método que vamos a aplicar a continuación, recibe el nombre de *RFE* (Recursive Feature Elimination). Consiste en ordenar las variables input por su poder predictivo (R^2 en el caso de variables objetivo numéricas) y, posteriormente crear modelos anidados en los que se vayan considerando modelos con una mayor cantidad de variables siguiendo ese orden.

Empezamos observando esa ordenación, para lo cual recurrimos a la función *filterVarImp()* de la librería *caret*:

▼ Mostrar el código

```
# El 4 es la posición en la que se ubica la variable IMC
salida<-filterVarImp(x = data_rg_train[,-4], y = data_rg_train$IMC, nonpara = TRUE)
ranking<-sort(apply(salida, 1, mean), decreasing =T)

# Para ajustar el margen inferior del gráfico y que así quepan los nombres
par(mar=c(8.1, 4.1, 4.1, 2.1))
barplot(ranking, las=2)
```



▼ Mostrar el código

```
# Vuelvo a poner los márgenes por defecto
par(mar=c(5.1, 4.1, 4.1, 2.1))
```

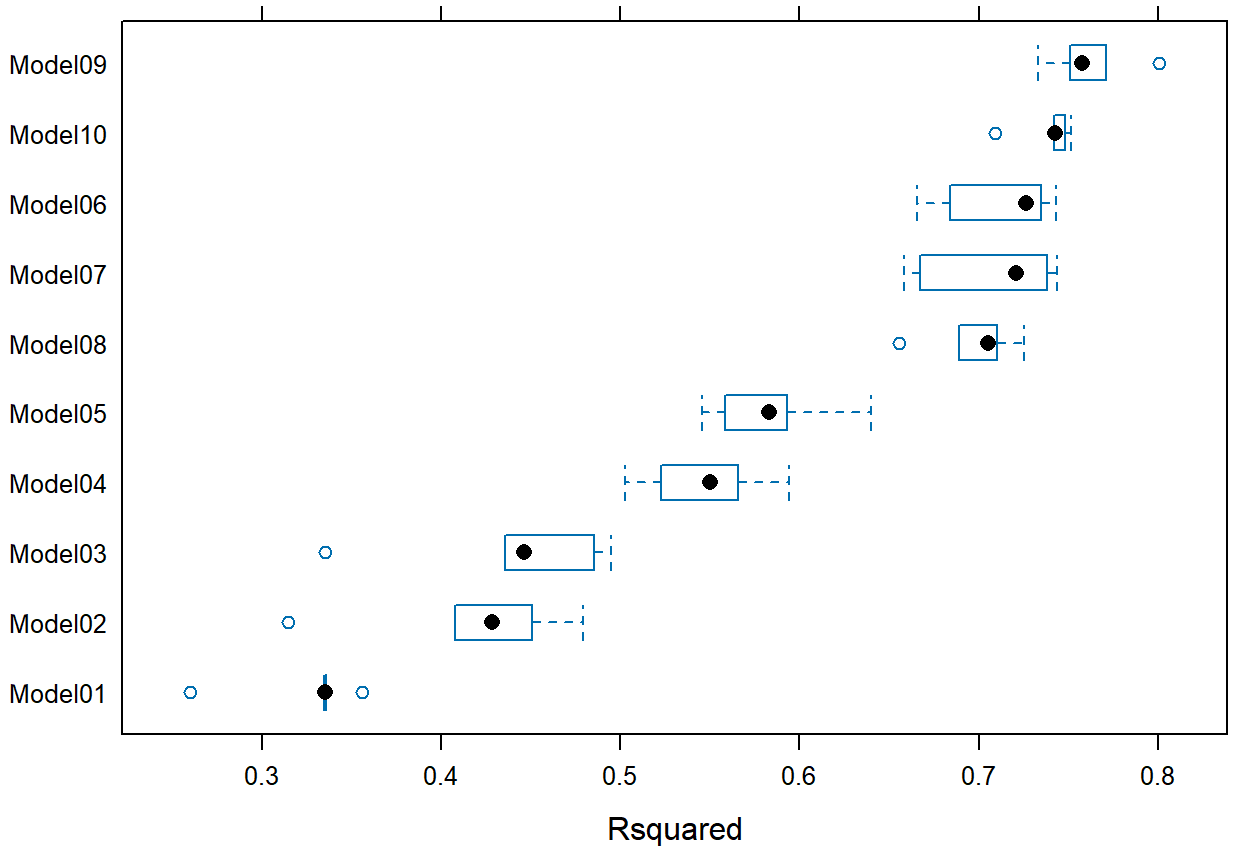
Observamos que las 3 primeras variables más importantes son las vistas en anteriores ocasiones. Así mismo, parece que la variable “NumeroComidas” carece completamente de potencial predictivo.

A continuación, a través de un bucle, vamos a ir construyendo modelos secuencialmente más complejos. Desgraciadamente, la función no permite construir modelos KNN con una única variable predictora, por lo que partiremos desde 2 variables, hasta todas. De nuevo, evitamos repetir la validación cruzada para aligerar el proceso:

▼ Mostrar el código

```
vcrTodosModelos<-list()
for (i in 1:(length(ranking)-1)){
set.seed(12345)
vcrTodosModelos[[i]] <- train(y=data_rg_train$IMC, x = data_rg_train[,names(ranking)[1:(i+1)]],
                  method = "kknn",
                  trControl = trainControl(method="cv", number = 5),
                  metric="Rsquared",
                  tuneGrid = expand.grid(kmax=6, distance=1, kernel = "rectangular"))
```

```
}
bwplot(resamples(vcrTodosModelos),metric=c("Rsquared"))
```



▼ Mostrar el código

```
summary(resamples(vcrTodosModelos),metric=c("Rsquared"))
```

```
Call:
summary.resamples(object = resamples(vcrTodosModelos), metric = c("Rsquared"))

Models: Model01, Model02, Model03, Model04, Model05, Model06, Model07, Model08, Model09, Model10
Number of resamples: 5

Rsquared
             Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
Model01 0.2599015 0.3350070 0.3353024 0.3244624 0.3360301 0.3560708    0
Model02 0.3147379 0.4080233 0.4286554 0.4164064 0.4510988 0.4795164    0
Model03 0.3351291 0.4358756 0.4461752 0.4394511 0.4851918 0.4948839    0
Model04 0.5026024 0.5226689 0.5502017 0.5470681 0.5654917 0.5943758    0
Model05 0.5458336 0.5587030 0.5830138 0.5840615 0.5930578 0.6396996    0
Model06 0.6658422 0.6841001 0.7260995 0.7108249 0.7349523 0.7431304    0
Model07 0.6583750 0.6673391 0.7207580 0.7056594 0.7381900 0.7436350    0
Model08 0.6553394 0.6887110 0.7054274 0.6970004 0.7102294 0.7252946    0
```

```
Model09 0.7328114 0.7510017 0.7576097 0.7625966 0.7708192 0.8007408    0
Model10 0.7091762 0.7417145 0.7424752 0.7385842 0.7478924 0.7516629    0
```

Hemos indicado el valor de k y el tipo de distancia que maximizaban el R^2 en el análisis preliminar. A continuación, podemos observar que los resultados mejoran a medida que se aumenta el número de variables, aunque no se trata de una relación completamente lineal. Por ejemplo, el Model09 resulta mejor que el Model10 y el Model06 es mejor que el Model08, lo que nos indica qué variables pueden resultar más/menos útiles. Es importante destacar que **el nombre de los modelos indica una variable menos** de la realidad. Observando la ubicación de las cajas y su variabilidad, concluimos que lo mejor podría ser el modelo 9, por lo que solo tendríamos que destarcar la última variable, que era Número de Comidas. No obstante lo anterior, también podría plantearse trabajar con el Model06 si se desea un modelo que cuente con menos variables input, pues no resulta excesivamente peor.

Una vez determinada la mejor combinación de variables, vamos a llevar a cabo una parametrización más fina, con valores más pequeños de k, pues ya vimos que no era necesario trabajar con valores grandes. Este proceso lo realizamos descartando las variables "inútiles" que nos haya indicado el modelo anterior; por eso, indicamos 9 en el número de columnas de la matriz x:

▼ Mostrar el código

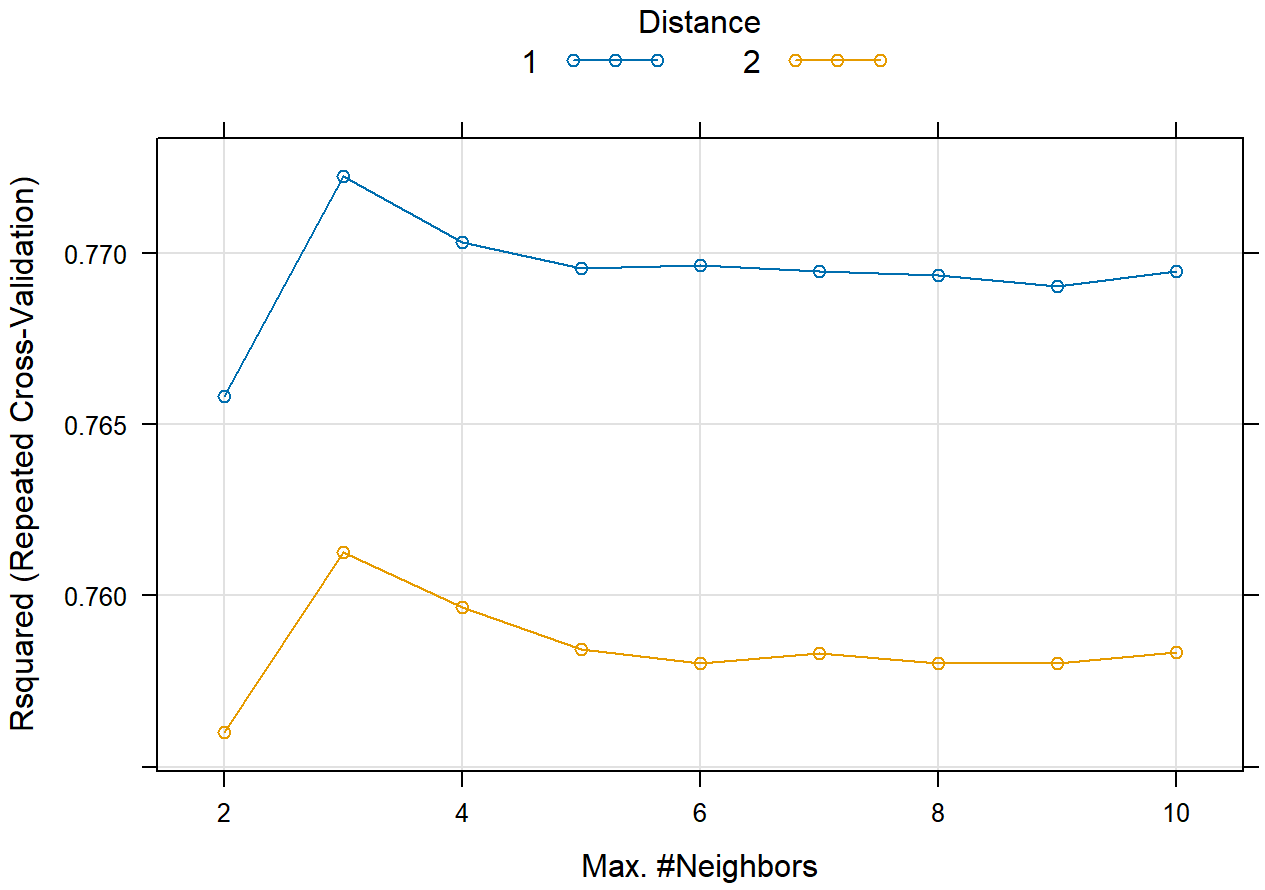
```
set.seed(12345)
knn_finetune <- train(y=data_rg_train$IMC, x = data_rg_train[,names(ranking)[1:(9+1)]],
                  method = "kknn",
                  trControl = trainControl(method="repeatedcv", number = 5, repeats=10),
                  tuneGrid = expand.grid(kmax=floor(seq.int(2,10,length.out=10)), distance=c(1,2)
knn_finetune
```

```
k-Nearest Neighbors

1665 samples
  10 predictor

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 10 times)
Summary of sample sizes: 1332, 1332, 1332, 1333, 1331, 1333, ...
Resampling results across tuning parameters:

  kmax  distance  RMSE      Rsquared   MAE
   2    1         3.959071  0.7658281  2.403472
   2    2         4.043083  0.7560095  2.476242
   3    1         3.873896  0.7722211  2.447000
   3    2         3.969227  0.7612582  2.514423
   4    1         3.883947  0.7703159  2.486307
   4    2         3.978571  0.7596581  2.545611
   5    1         3.888462  0.7695431  2.507572
   5    2         3.986193  0.7584250  2.565505
   6    1         3.887160  0.7696319  2.513673
   6    2         3.988295  0.7580242  2.574264
   7    1         3.888655  0.7694706  2.515869
```

```
 7     2         3.986418  0.7583144  2.570197
 8     1         3.889752  0.7693488  2.516361
 8     2         3.988295  0.7580242  2.574264
 9     1         3.891650  0.7690308  2.519817
 9     2         3.988295  0.7580242  2.574264
10     1         3.888655  0.7694706  2.515869
10     2         3.986027  0.7583542  2.570130

Tuning parameter 'kernel' was held constant at a value of rectangular
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were kmax = 3, distance = 1 and kernel
 = rectangular.
```

▼ Mostrar el código

```
plot(knn_finetune,metric=c("Rsquared"))
```



De nuevo comprobamos que es preferible trabajar con la distancia de Manhattan sobre estos datos. Verificamos que el k óptimo en este caso es 3, aunque no existen grandes diferencias entre los modelos considerados.

Destacamos que esta fase sí se ha realizado con validación cruzada repetida, pues se trata de una búsqueda más concreta.

Análisis final del modelo

Una vez determinadas las variables y los parámetros óptimos, vamos a construir el modelo "final", para poder obtener la estimación realista de la calidad a partir de los datos de prueba:

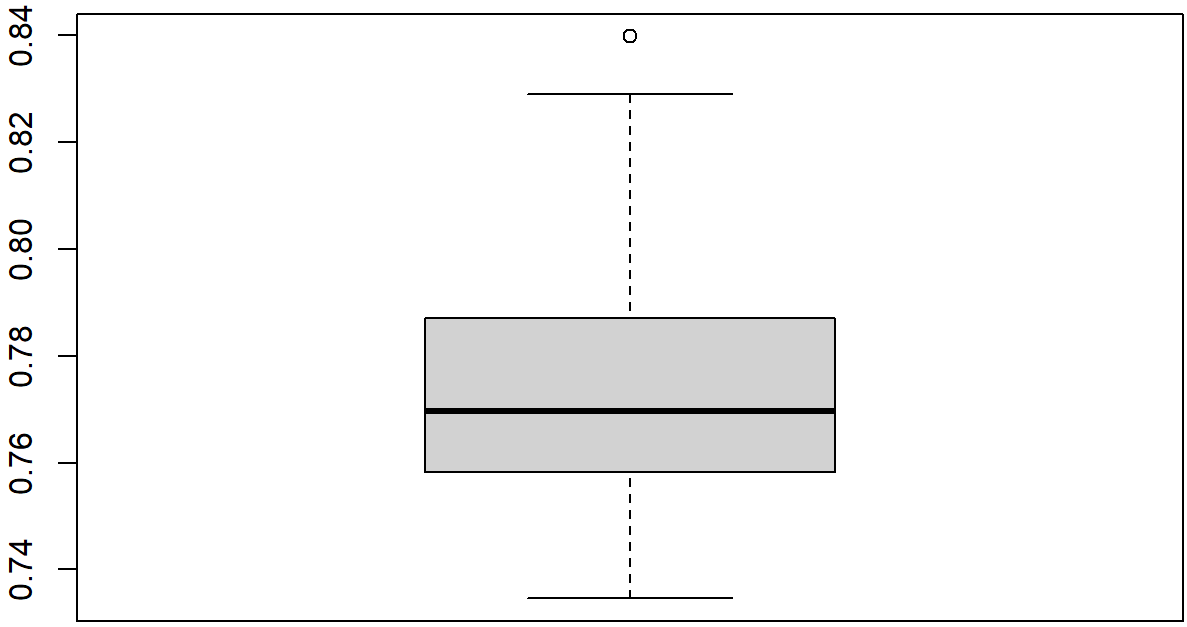
▼ Mostrar el código

```
modeloFinal <- kknn(IMC~.-NumeroComidas, data_rg_train, data_rg_test, distance = 1, k=4, kernel =
R2(modeloFinal$fitted.values,data_rg_test$IMC)
```

```
[1] 0.7476373
```

▼ Mostrar el código

```
boxplot(knn_finetune$resample$Rsquared)
```

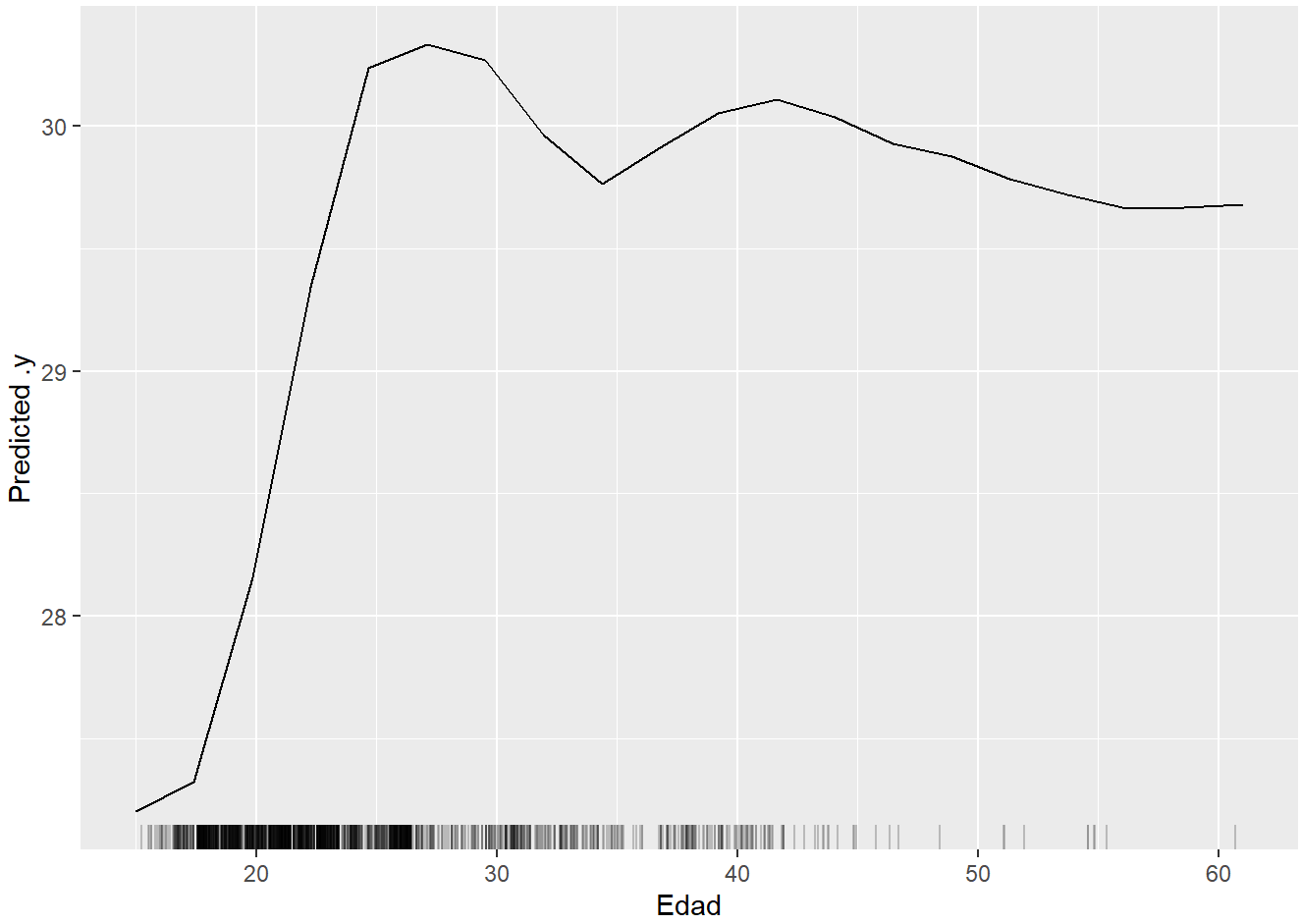


Observamos que hemos conseguido un R^2 relativamente alto, gracias a que este modelo es más flexible (aunque también menos interpretable). Dado que no sería correcto evaluar la estabilidad a partir de los datos de entrenamiento, hemos obtenido el diagrama de cajas para los resultados de la validación cruzada. Observamos que el R^2 tiene un recorrido intercuartílico pequeño (la caja se mueve entre 0.76 y 0.79 aproximadamente), por lo que se trata de un modelo relativamente estable.

Para finalizar, vamos a recurrir a los gráficos de dependencia parcial (PDP) para poder obtener algo de información sobre el efecto de las variables input sobre la objetivo. Por como está diseñada la librería *iml* (interpretable machine learning), se debe utilizar el objeto generado con la función *train()* de *caret*. Optamos por algunas de las variables más importantes/interesantes (de una en una o en par):

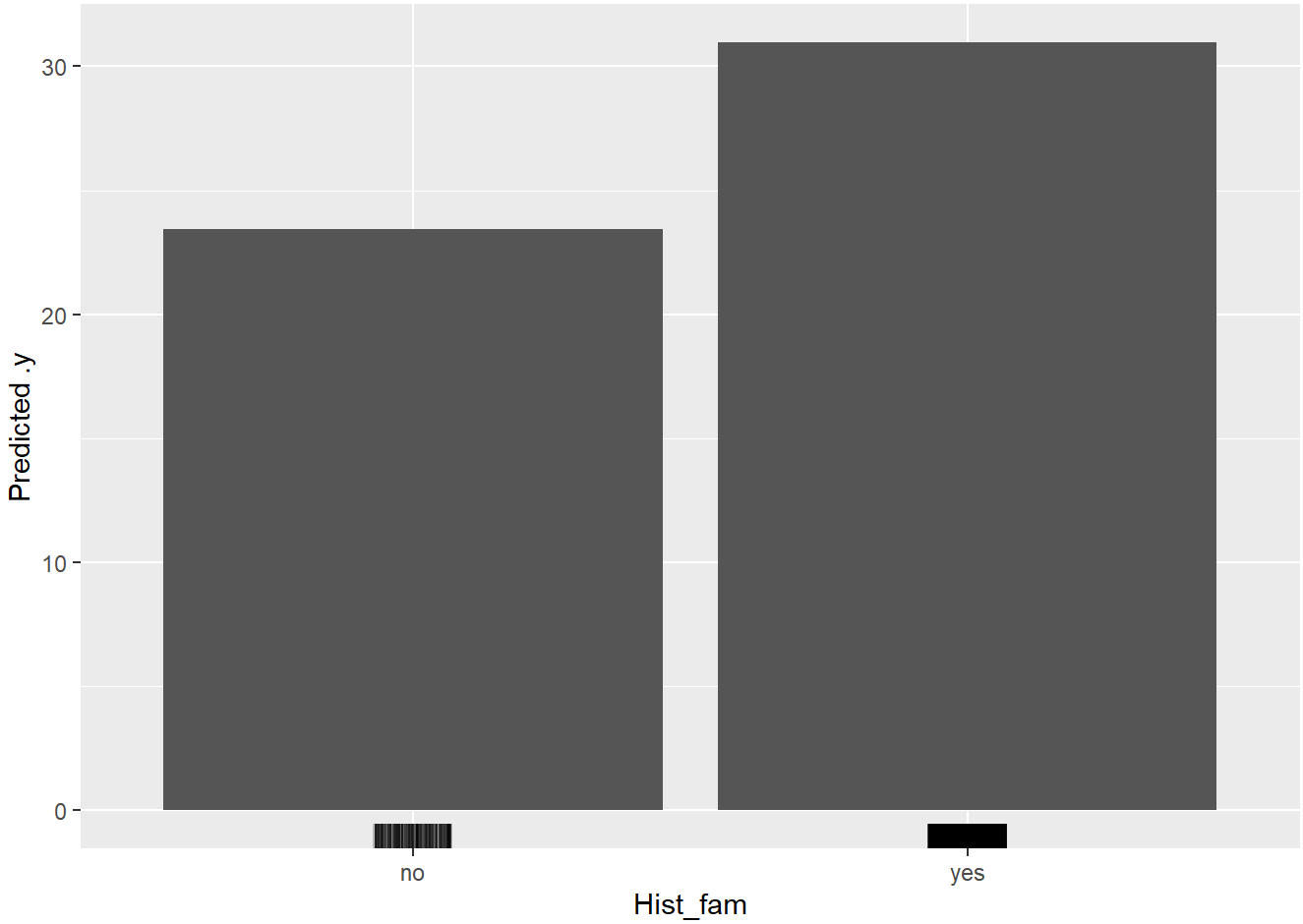
▼ Mostrar el código

```
predictor <- Predictor$new(knn_finetune, data = data_rg_train, y = data_rg_train$IMC)
pdp <- FeatureEffect$new(predictor, feature = "Edad", method="pdp")
pdp$plot()
```



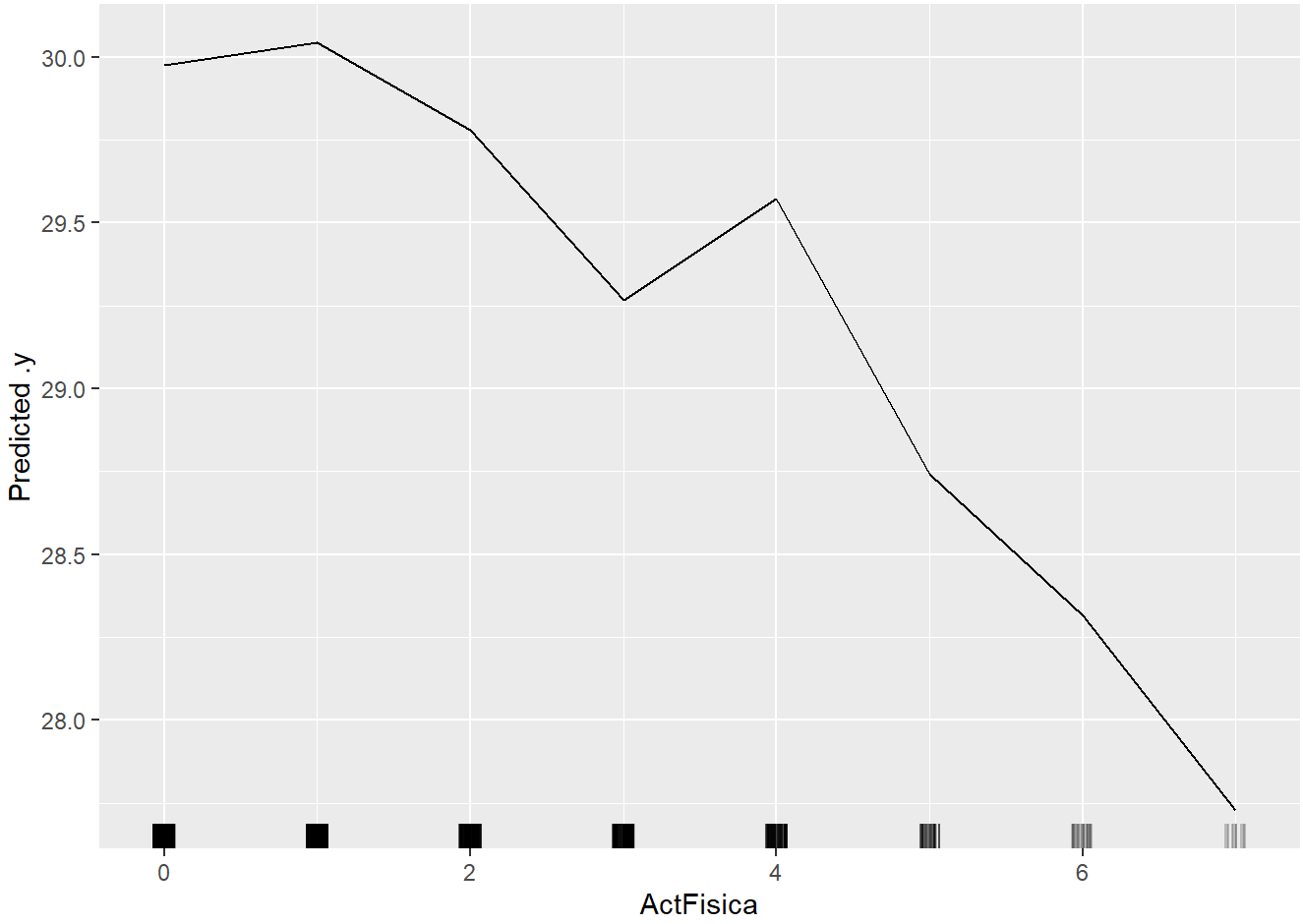
▼ Mostrar el código

```
pdp2 <- FeatureEffect$new(predictor, feature = "Hist_fam", method="pdp")
pdp2$plot()
```



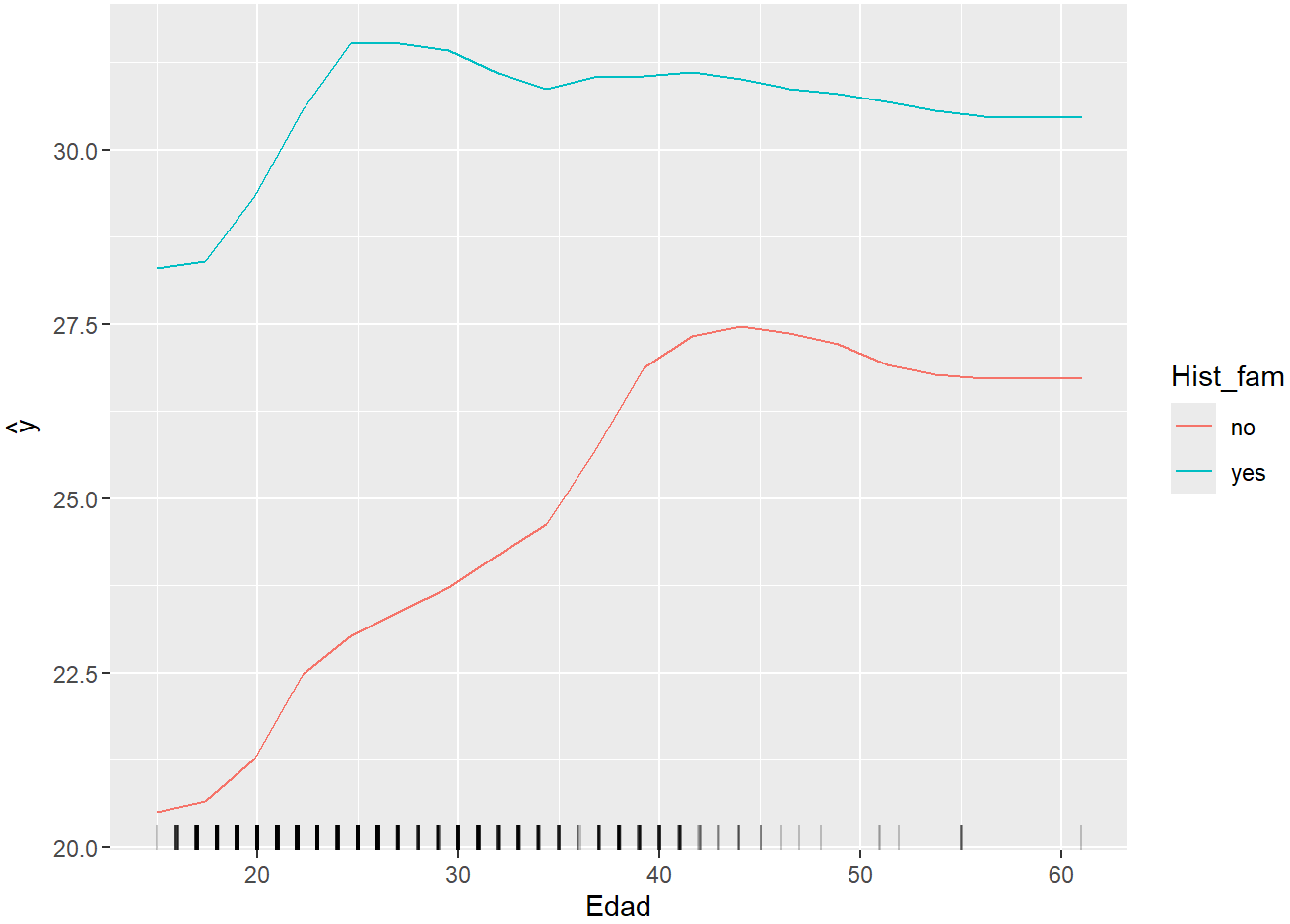
▼ Mostrar el código

```
pdp3 <- FeatureEffect$new(predictor, feature = "ActFisica", method="pdp", grid.size = 8) # Ponemo
pdp3$plot()
```



▼ Mostrar el código

```
pdp4 <- FeatureEffect$new(predictor, feature = c("Hist_fam", "Edad"), method="pdp")
pdp4$plot()
```



Se han construido distintos gráficos, lo que permite extraer las siguientes conclusiones:

- El efecto de la edad en el IMC no es lineal, ni monótono. Se observa como crece hasta los 27 años, luego decrece hasta los 35 y se estanca a partir de los 40.
- En líneas generales, las personas con historial familiar de sobrepeso tienen un IMC 7.5 puntos superior.
- El IMC no decrece de manera lineal con el aumento de la actividad física. De hecho, a penas existe diferencia entre el 0 y el 1 y, sorprendentemente, es ligeramente superior en 4 que en 3.
- Existe cierta interacción entre el historial familiar y la edad, pues las dos gráficas, aunque similares, no son iguales. De no existir interacción, deberían tener la forma de la edad cuando se analiza de manera univariante, pero con un cambio de escala equivalente a la diferencia observada en la variable *Hist_fam*. Dentro de las personas que no tienen historial familiar de sobrepeso, el IMC crece linealmente con la edad hasta los 40 años, pero no es así entre las personas que sí tienen dicho historial. Así mismo, se observa claramente que se trata de dos variables muy importantes pues existe una diferencia de 13 puntos aproximadamente en el IMC entre los extremos.

KNN para clasificación

En este apartado vamos a adaptar los procesos anterior al caso de clasificación, aunque mucha de la información estará repetida. Para ello, intentamos predecir si los individuos tienen o no sobrepeso. Esta

variable está codificada como “1” (sí se tiene sobrepeso) y “0” (no se tiene sobrepeso). Para que las funciones que aplicaremos en este documento funcionen correctamente, es recomendable codificar dicha variable. Para ello, la opción mas cómoda consiste en recurrir a la función *make.names()* y volverlo a convertir en factor.

A continuación, observamos el reparto de la misma:

▼ Mostrar el código

```
datosClasif$Sobrepeso<-as.factor(make.names(datosClasif$Sobrepeso))
table(datosClasif$Sobrepeso)/nrow(datosClasif)
```

```
       X0        X1 
0.2694899 0.7305101 
```

Antes de comenzar con el proceso de modelización, procedemos a llevar a cabo la partición entrenamiento-prueba del otro conjunto de datos:

▼ Mostrar el código

```
set.seed(12345)
  trainIndex <- createDataPartition(datosClasif$Sobrepeso, p=0.8, list=FALSE)
data_clasif_train <- datosClasif[trainIndex,]
data_clasif_test <- datosClasif[-trainIndex,]
```

Continuamos mostrando cómo obtener modelos KNN para clasificación, es decir, para variables dependientes cualitativas (no hay diferencias entre el número de valores que pueda tomar ni si existe orden entre ellos). De nuevo, el principal parámetro a fijar es el número de vecinos que se va a considerar. La versión más clásica de los modelos KNN pondera por igual la información de todos los vecinos (es decir se realiza una media aritmética), pero esto se podría modificar para darle más peso a las observaciones más próximas. La función *kknn()* que vamos a utilizar permite esta opción mediante el parámetro *kernel* pero vamos a ignorarlo en este curso para simplificar. Se puede consultar la ayuda de la función si se desea información adicional.

Vamos a construir un primer modelo con 5 vecinos y después intentaremos mejorar los resultados modificando los parámetros. Cabe destacar que la función que vamos a utilizar estandariza internamente las variables y convierte en *dummy* las cualitativas, por lo que no es necesario ningún preprocesamiento:

▼ Mostrar el código

```
modelo1 <- kknn(Sobrepeso~., data_clasif_train, data_clasif_test, distance = 2, k=5, kernel = "re
head(modelo1$D)
```

```
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 1.7284286 1.812601 2.020127 2.509156 2.583665
[2,] 0.7074544 1.119137 1.309950 1.325853 1.416107
[3,] 0.6951322 1.275007 1.350739 1.414071 1.684096
[4,] 0.7758334 1.645155 1.996432 2.156864 2.348415
```

```
[5,] 1.2746188 1.689638 1.780278 1.808419 1.858529
[6,] 1.8858515 1.904865 1.931268 2.560959 2.571658
```

▼ Mostrar el código

```
head(modelo1$CL)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] "X0" "X1" "X0" "X0" "X0"
[2,] "X1" "X1" "X1" "X1" "X1"
[3,] "X1" "X1" "X0" "X1" "X1"
[4,] "X0" "X0" "X1" "X0" "X0"
[5,] "X1" "X1" "X1" "X1" "X1"
[6,] "X0" "X0" "X0" "X0" "X0"
```

▼ Mostrar el código

```
head(modelo1$prob)
```

```
      X0  X1
[1,] 0.8 0.2
[2,] 0.0 1.0
[3,] 0.2 0.8
[4,] 0.8 0.2
[5,] 0.0 1.0
[6,] 1.0 0.0
```

▼ Mostrar el código

```
head(modelo1$fitted.values)
```

```
[1] X0 X1 X1 X0 X1 X0
Levels: X0 X1
```

Esta función requiere que le indiquemos la fórmula (es decir, variable objetivo y las input), además de los conjuntos de datos de entrenamiento y prueba, el número de vecinos (que es recomendable que sea impar), el tipo de ponderación (rectangular implica mismo peso para todos los vecinos) y el tipo de distancia. De manera genérica, se considera la distancia Minkowski, que tiene como caso particular la distancia euclídea cuando *distance=2*. Así mismo, si *distance=1*, se estaría utilizando la distancia de *Manhattan*, que es preferible cuando existen muchas variables input y/o existen datos atípicos.

Para entender mejor el funcionamiento de este método, hemos pedido que nos muestre parte de las matrices D y CL, que contienen las distancias a los $k = 5$ vecinos más próximos de las observaciones en el conjunto de datos de prueba, y los valores de la variable objetivo, respectivamente. Cabe recordar que esos vecinos más próximos se seleccionan de entre las observaciones del conjunto de datos de entrenamiento. Finalmente, hemos pedido los valores predichos (tanto las probabilidades, como la clasificación); podemos verificar que las primeras resultan de hacer la media aritmética de las filas de la matriz CL.

A continuación, podemos calcular la calidad del modelo construido en prueba (cabe destacar que, en este punto, se debería adaptar la métrica de calidad al tipo de datos concretos, es decir, se puede optar por AUC, kappa, kappa ponderado, tasa de acierto, etc.). Optamos por el AUC pairwise por su versatilidad (pues en el caso binario es el AUC clásico):

▼ Mostrar el código

```
pROC::multiclass.roc(data_clasif_test$Sobrepeso, modelo1$prob)
```

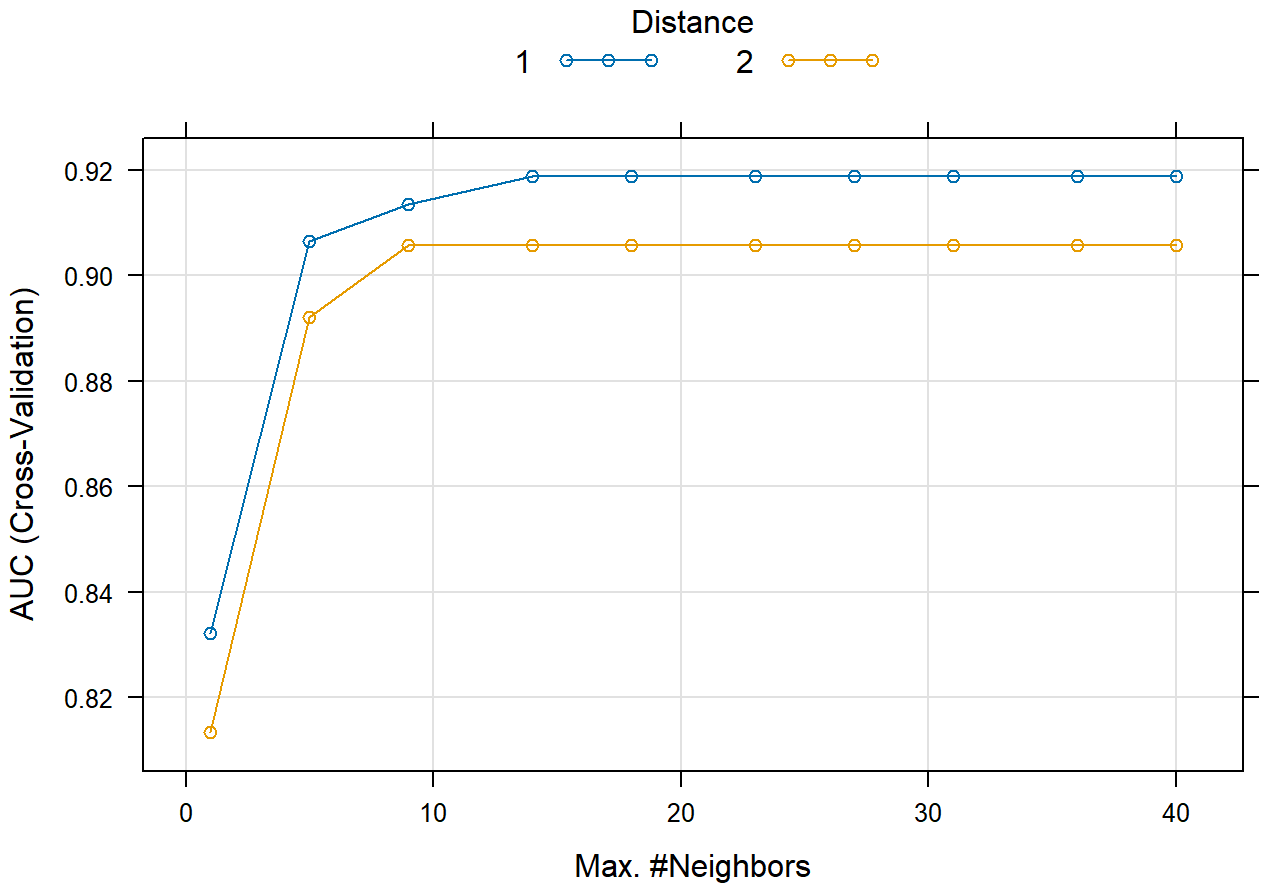
```
Call:
multiclass.roc.default(response = data_clasif_test$Sobrepeso,     predictor = modelo1$prob)

Data: multivariate predictor modelo1$prob with 2 levels of data_clasif_test$Sobrepeso: X0, X1.
Multi-class area under the curve: 0.8604
```

Desgraciadamente, no es posible obtener una estimación realista de la calidad en entrenamiento puesto que la propia "bolsa" donde se buscan los vecinos contiene a la observación y eso hace que el resultado sea optimista por naturaleza (por ejemplo, si se pone *k=1* el modelo sería perfecto). No obstante, desde el punto de vista de los datos de prueba, parece un modelo con relativamente buena capacidad predictiva.

Para poder determinar el valor óptimo de k y si es mejor la distancia euclídea o la de Manhattan, podemos recurrir a la validación cruzada (no la hacemos repetida pues se trata de un proceso costoso computacionalmente hablando), a través de la funcion *train()* de la librería caret. En esta ocasión, en lugar de indicar directamente la fórmula a aplicar, separamos la variables objetivo (*y*) y la matriz que contiene la información de las variables input (*x*):

▼ Mostrar el código

```
set.seed(12345)
# En la posición 4 está la variable objetivo
knn_tuneTodo <- train(y=data_clasif_train$Sobrepeso, x = data_clasif_train[,-4],
                  method = "kknn",
                  trControl = trainControl(method="cv", number = 5, summaryFunction=multiClassSum
                  metric="AUC",
                  tuneGrid = expand.grid(kmax=floor(seq.int(1,sqrt(nrow(data_rg_train)),length.ou
plot(knn_tuneTodo,metric=c("AUC"))
```



Algunos autores recomiendan como k genérico utilizar la raíz cuadrada del número de observaciones en entrenamiento. Dado que esto puede resultar excesivo, hemos evaluado una parrilla de valores, que van desde 2 hasta esa cantidad. Vemos que, a partir de $k = 15$ no se observan diferencias en el AUC, por lo que no es necesario un número de vecinos tan elevado para estos datos. Es importante destacar que, en el caso de que el mejor valor se observe en alguno de los extremos, sería recomendable aumentar por ese lado la rejilla de valores.

En cuanto al tipo de distancia, parece preferible recurrir a la distancia de Manhattan.

Hemos llevado una búsqueda de parámetros inicial para encontrar la zona más adecuada de k donde buscar. No obstante, como ya se mencionó en la teoría, los modelos KNN están fuertemente afectados por variables input sin poder predictivo, por lo que a continuación estudiamos cómo seleccionar variables.

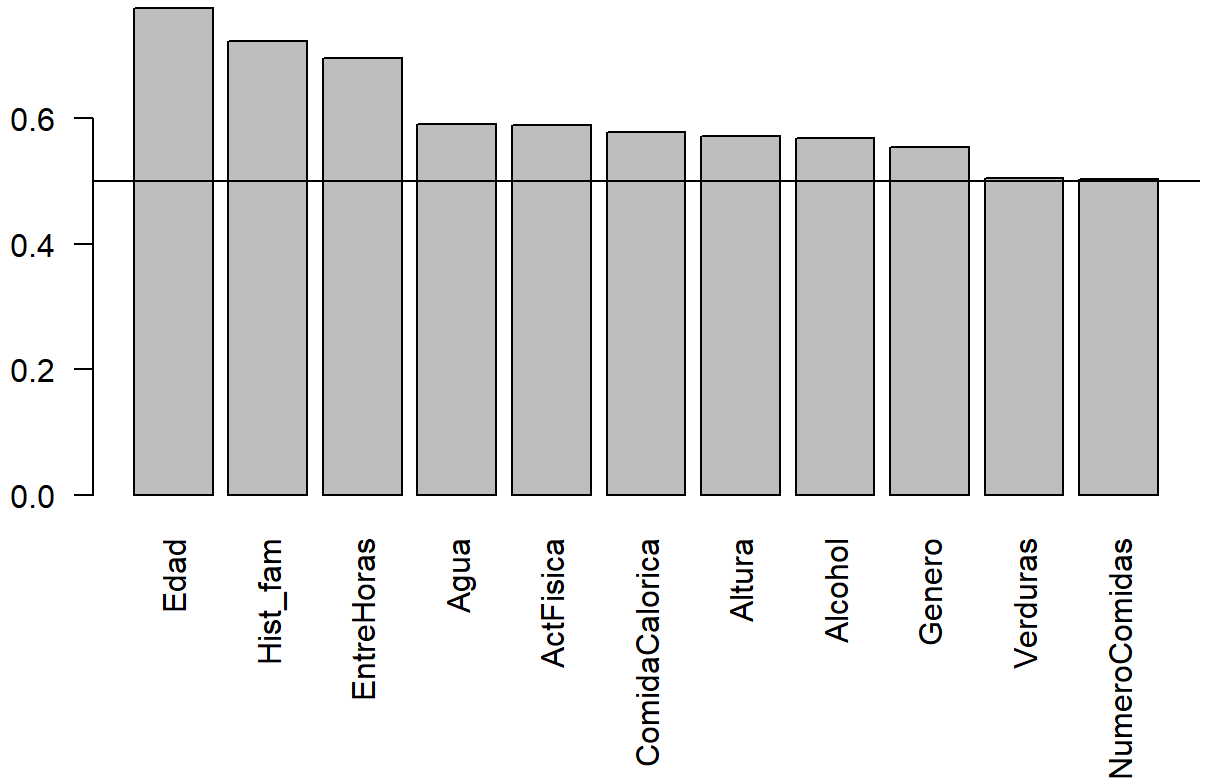
Selección de variables

El método que vamos a aplicar a continuación, recibe el nombre de *RFE* (Recursive Feature Elimination). Consiste en ordenar las variables input por su poder predictivo (AUC en el caso de variables objetivo categóricas) y, posteriormente crear modelos anidados en los que se vayan considerando modelos con una mayor cantidad de variables siguiendo ese orden.

Empezamos observando esa ordenación, para lo cual recurrimos a la función *filterVarImp()* de la librería *caret*:

▼ Mostrar el código

```
# El 4 es la posición en la que se ubica la variable IMC
salida<-filterVarImp(x = data_clasif_train[,-4], y = data_clasif_train$Sobrepeso, nonpara = TRUE)
ranking<-sort(apply(salida, 1, mean), decreasing =T)

# Para ajustar el margen inferior del gráfico y que así quepan los nombres
par(mar=c(8.1, 4.1, 4.1, 2.1))
barplot(ranking, las=2)
abline(h=0.5)
```



▼ Mostrar el código

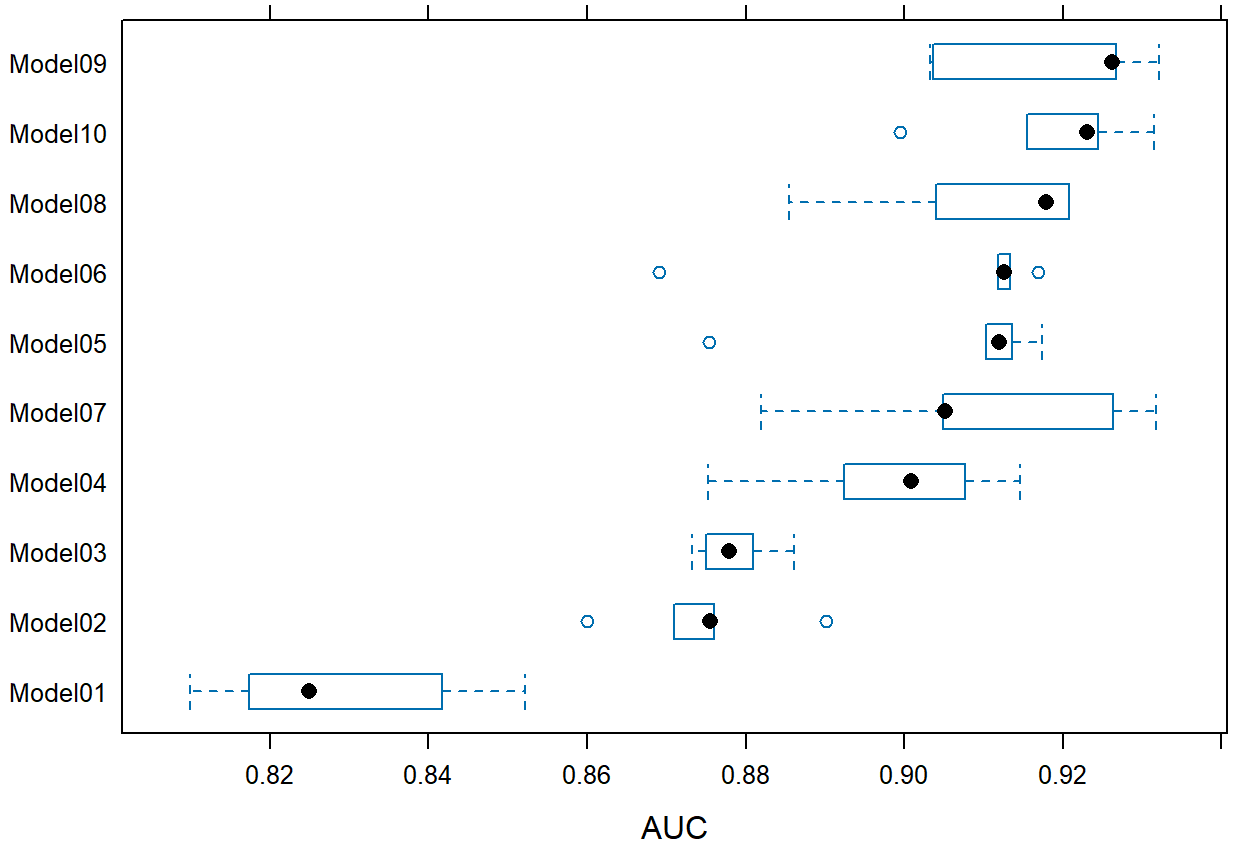
```
# Vuelvo a poner los márgenes por defecto
par(mar=c(5.1, 4.1, 4.1, 2.1))
```

Observamos que las 3 primeras variables más importantes son las vistas en anteriores ocasiones. Así mismo, parece que la variable "NumeroComidas" carece completamente de potencial predictivo, pues su AUC se sitúa sobre la línea de referencia ubicada en 0.5. Tampoco parece muy útil la variable *Verduras*.

A continuación, a través de un bucle, vamos a ir construyendo modelos secuencialmente más complejos. Desgraciadamente, la función no permite construir modelos KNN con una única variable predictora, por lo que partiremos desde 2 variables, hasta todas. De nuevo, evitamos repetir la validación cruzada para aligerar el proceso:

▼ Mostrar el código

```
vcrTodosModelos<-list()
for (i in 1:(length(ranking)-1)){
set.seed(12345)
vcrTodosModelos[[i]] <- train(y=data_clasif_train$Sobrepeso, x = data_clasif_train[,names(ranking
                  method = "kknn",
                  trControl = trainControl(method="cv", number = 5, summaryFunction=multiClassSum
                  metric="AUC",
                  tuneGrid = expand.grid(kmax=15, distance=1, kernel = "rectangular"))
}
bwplot(resamples(vcrTodosModelos),metric=c("AUC"))
```



▼ Mostrar el código

```
summary(resamples(vcrTodosModelos),metric=c("AUC"))
```

```
Call:
summary.resamples(object = resamples(vcrTodosModelos), metric = c("AUC"))

Models: Model01, Model02, Model03, Model04, Model05, Model06, Model07, Model08, Model09, Model10
Number of resamples: 5
```

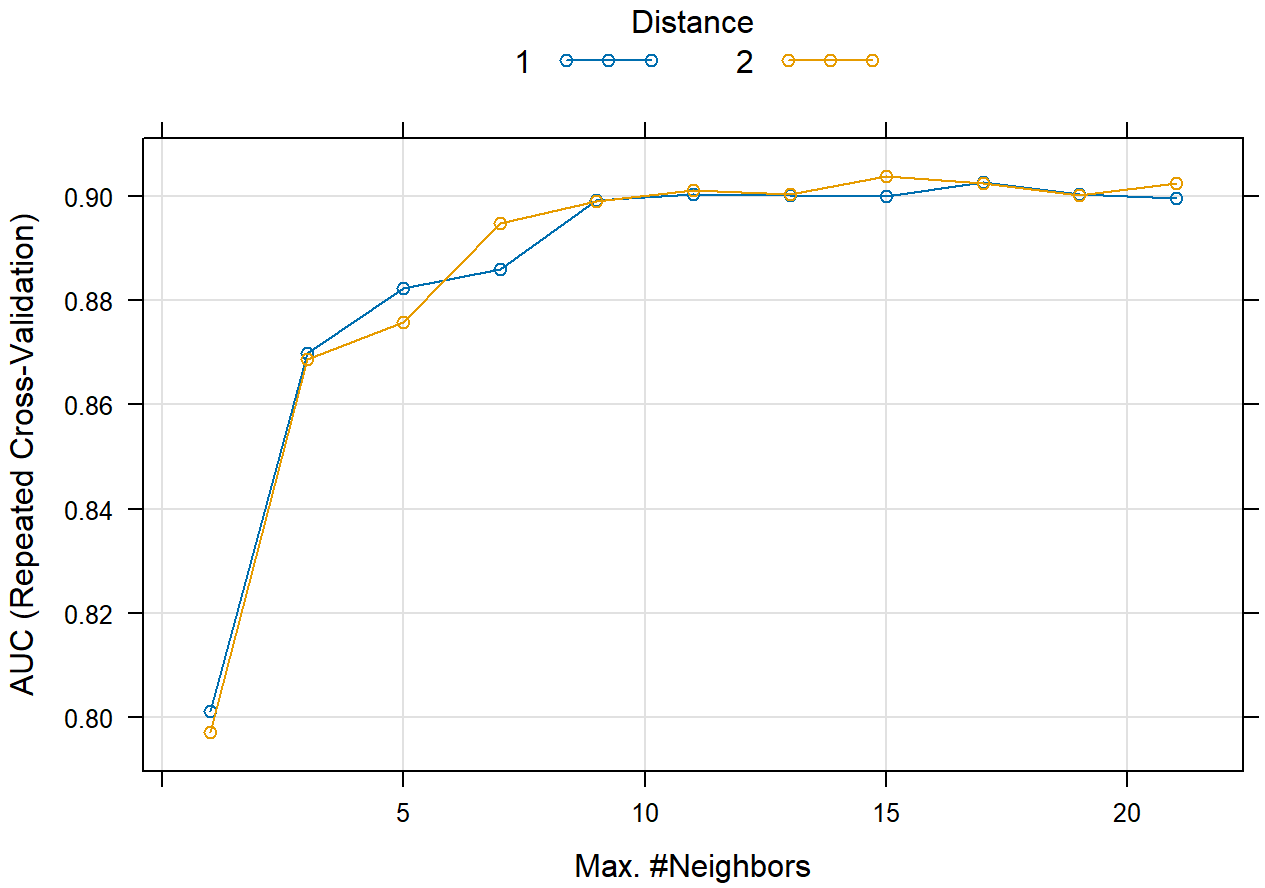
```
AUC
             Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
Model01 0.8098904 0.8173983 0.8249642 0.8292343 0.8417010 0.8522176    0
Model02 0.8600366 0.8709946 0.8755722 0.8745671 0.8760631 0.8901692    0
Model03 0.8732510 0.8749657 0.8779072 0.8786572 0.8810014 0.8861608    0
Model04 0.8752629 0.8923640 0.9009145 0.8981786 0.9077080 0.9146437    0
Model05 0.8753315 0.9102973 0.9120081 0.9057322 0.9136260 0.9173983    0
Model06 0.8690672 0.9118001 0.9125861 0.9047401 0.9133745 0.9168724    0
Model07 0.8819616 0.9048874 0.9051669 0.9100227 0.9262958 0.9318016    0
Model08 0.8855053 0.9040695 0.9178342 0.9098060 0.9207703 0.9208505    0
Model09 0.9032693 0.9036808 0.9262460 0.9184153 0.9266889 0.9321912    0
Model10 0.8994513 0.9154778 0.9230360 0.9187878 0.9244694 0.9315043    0
```

Hemos indicado el valor de k y el tipo de distancia que maximizaban el AUC en el análisis preliminar. A continuación, podemos observar que los resultados mejoran a medida que se aumenta el número de variables, aunque no se trata de una relación completamente lineal. Por ejemplo, el Model09 resulta mejor que el Model10 y el Model05 es mejor que el Model07, lo que nos indica qué variables pueden resultar más/menos útiles. Es importante destacar que **el nombre de los modelos indica una variable menos** de la realidad. En este caso, teniendo en cuenta sobre todo la variabilidad, una buena opción podría ser el Model05, pues, aunque su mediana sea ligeramente inferior, su caja se encuentra completamente incluída en el Model09. Sería un modelo con mucho menos sobreajuste.

Una vez determinada la mejor combinación de variables, vamos a llevar a cabo una parametrización más fina, con valores más específicos de k, pues ya vimos que no era necesario trabajar con valores tan grandes. Este proceso lo realizamos descartando las variables "inútiles" que nos haya indicado el modelo anterior; por eso, indicamos 5 en el número de columnas de la matriz x:

▼ Mostrar el código

```
set.seed(12345)
knn_finetune <- train(y=data_clasif_train$Sobrepeso, x = data_clasif_train[,names(ranking)[1:(5+1
                  method = "kknn", metric= "AUC",
                  trControl = trainControl(method="repeatedcv", number = 5, repeats=5, summaryFun
                  tuneGrid = expand.grid(kmax=floor(seq.int(1,21,length.out=11)), distance=c(1,2)
plot(knn_finetune,metric=c("AUC"))
```



Comprobamos que no existen grandes diferencias entre los dos tipos de distancias. No obstante, la euclídea resulta ligeramente mejor. Sí existen grandes diferencias en lo que se refiere a la selección del parámetro k, siendo el óptimo un valor de 15.

Destacamos que esta fase sí se ha realizado con validación cruzada repetida, pues se trata de una búsqueda más concreta.

Análisis final del modelo

Una vez determinadas las variables y los parámetros óptimos, vamos a construir el modelo "final", para poder obtener la estimación realista de la calidad a partir de los datos de prueba:

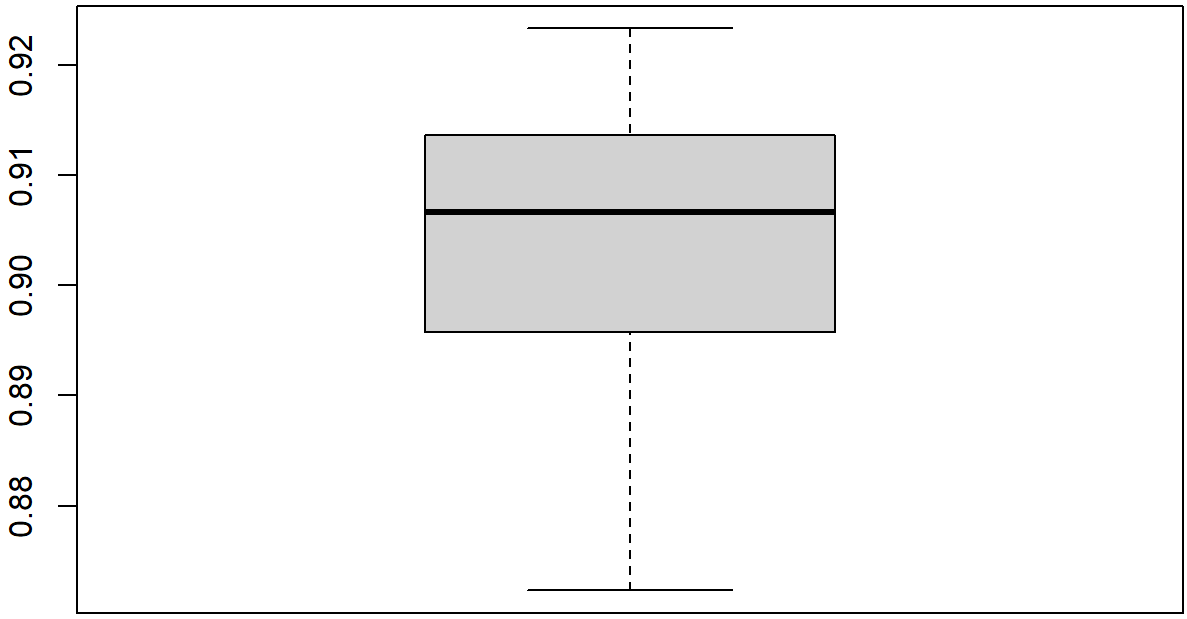
▼ Mostrar el código

```
modeloFinal <- kknn(Sobrepeso~Edad+Hist_fam+EntreHoras+Agua+ActFisica, data_clasif_train, data_cl
pROC::multiclass.roc(data_clasif_test$Sobrepeso, modelo1$prob)
```

```
Call:
multiclass.roc.default(response = data_clasif_test$Sobrepeso,     predictor = modelo1$prob)

Data: multivariate predictor modelo1$prob with 2 levels of data_clasif_test$Sobrepeso: X0, X1.
Multi-class area under the curve: 0.8604
```

▼ Mostrar el código

```
boxplot(knn_finetune$resample$AUC)
```



▼ Mostrar el código

```
cm_test<-confusionMatrix(table(modeloFinal$fitted.values,data_clasif_test$Sobrepeso))
cm_test$table
```

```
    X0  X1
X0  73  15
X1  39 288
```

▼ Mostrar el código

```
cm_test$overall[1:2]
```

```
 Accuracy     Kappa
0.8698795 0.6459044
```

▼ Mostrar el código

```
cm_test$byClass[1:2]
```

```
Sensitivity Specificity
  0.6517857   0.9504950
```

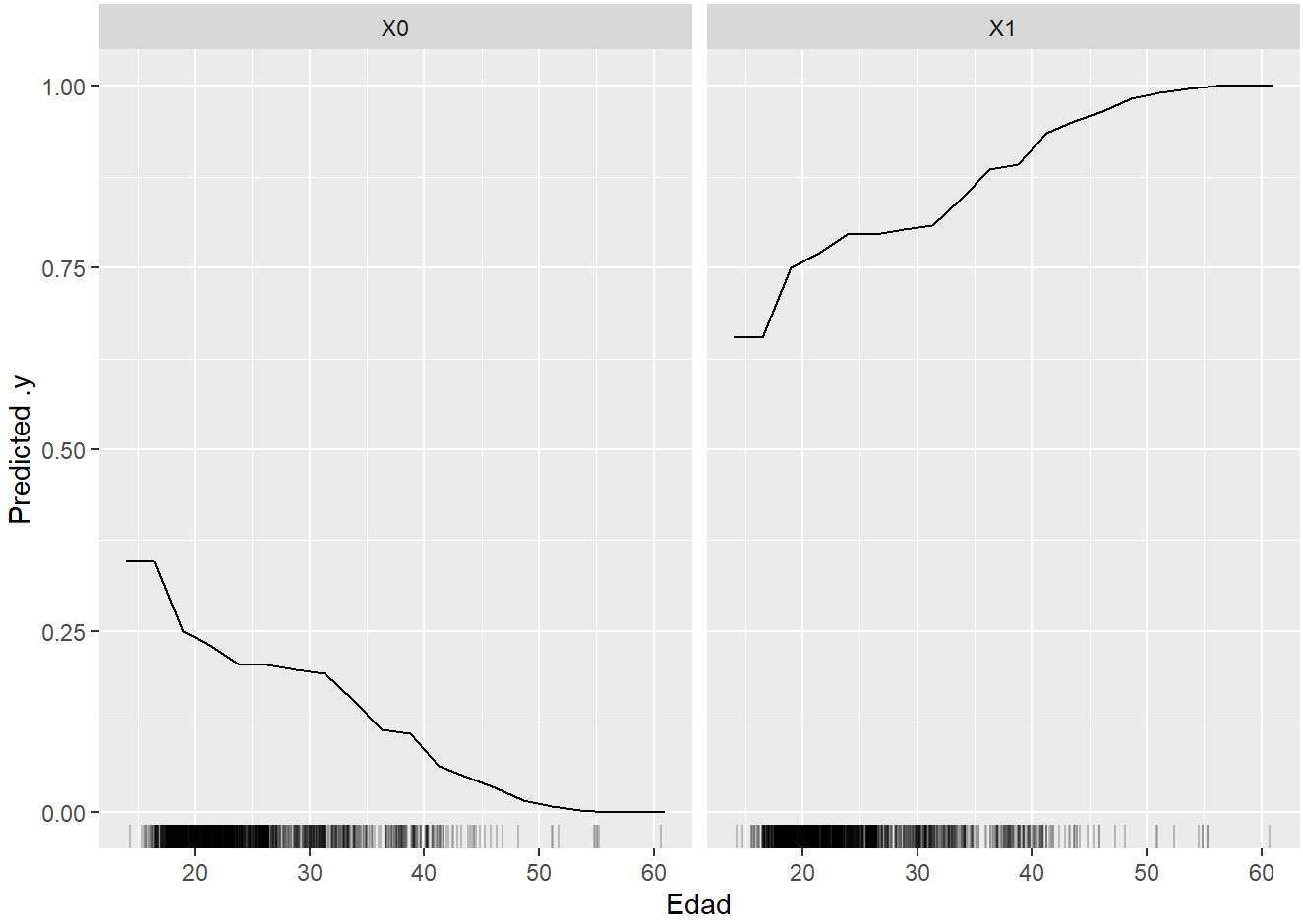
Observamos que hemos conseguido un AUC bastante alto, aunque en este caso no hay grandes diferencias con el modelo de regresión logística binaria. Dado que no sería correcto evaluar la estabilidad a partir de los datos de entrenamiento, hemos obtenido el diagrama de cajas para los resultados de la validación cruzada. Observamos que el AUC tiene un recorrido intercuartílico pequeño (la caja se mueve entre 0.89 y 0.91 aproximadamente), por lo que se trata de un modelo relativamente estable.

Hemos obtenido también las métricas asociadas con la matriz de confusión, lo que nos permite reafirmar la calidad del modelo, aunque está ciertamente desequilibrado en cuanto a sensibilidad y especificidad.

Para finalizar, vamos a recurrir a los gráficos de dependencia parcial (PDP) para poder obtener algo de información sobre el efecto de las variables input sobre la objetivo. Por como está diseñada la librería *iml* (interpretable machine learning), se debe utilizar el objeto generado con la función *train()* de *caret*. Optamos por algunas de las variables más importantes/interesantes (de una en una o en par):

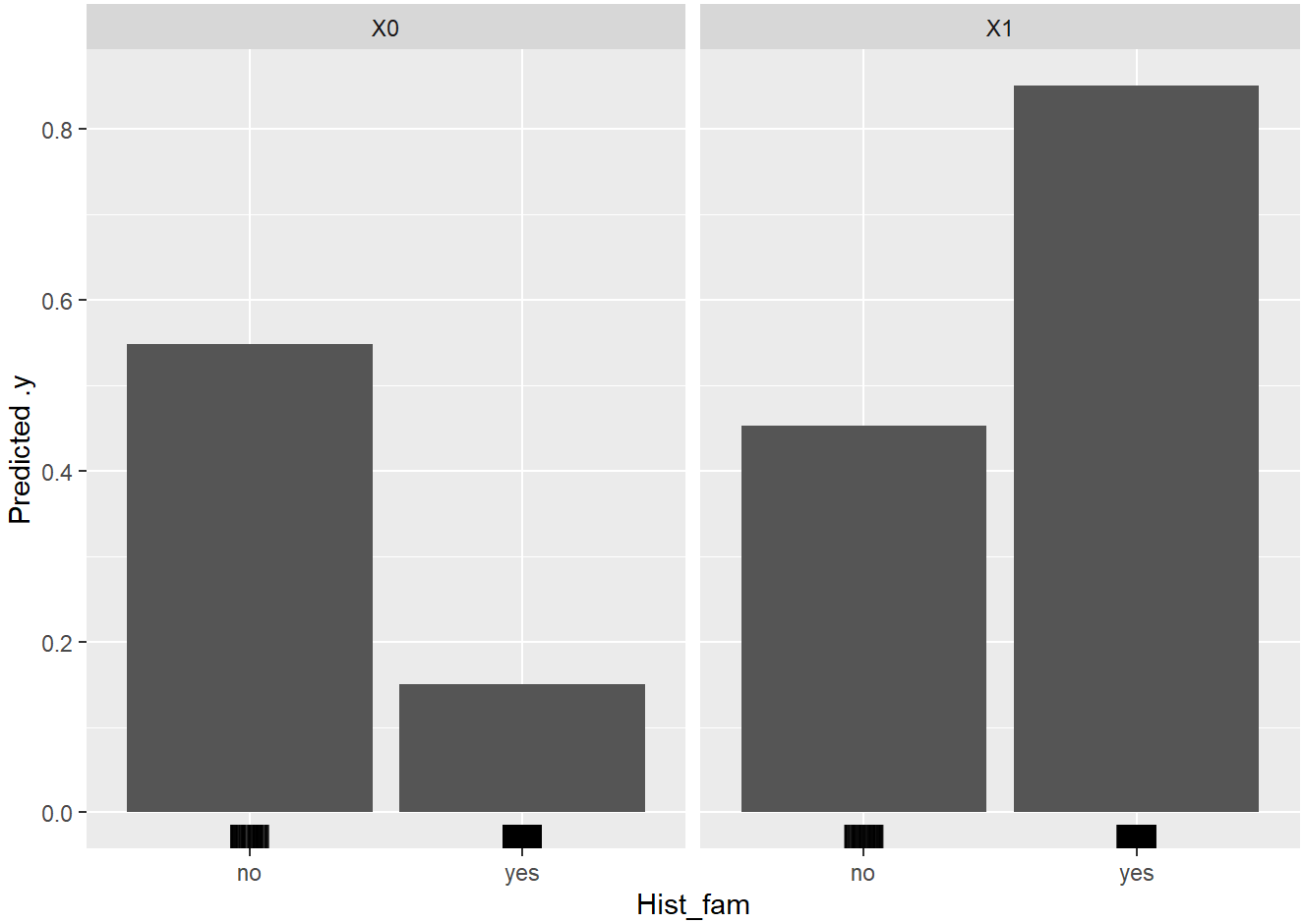
▼ Mostrar el código

```
predictor <- Predictor$new(knn_finetune, data = data_clasif_train, y = data_clasif_train$Sobrepes
pdp <- FeatureEffect$new(predictor, feature = "Edad", method="pdp")
pdp$plot()
```



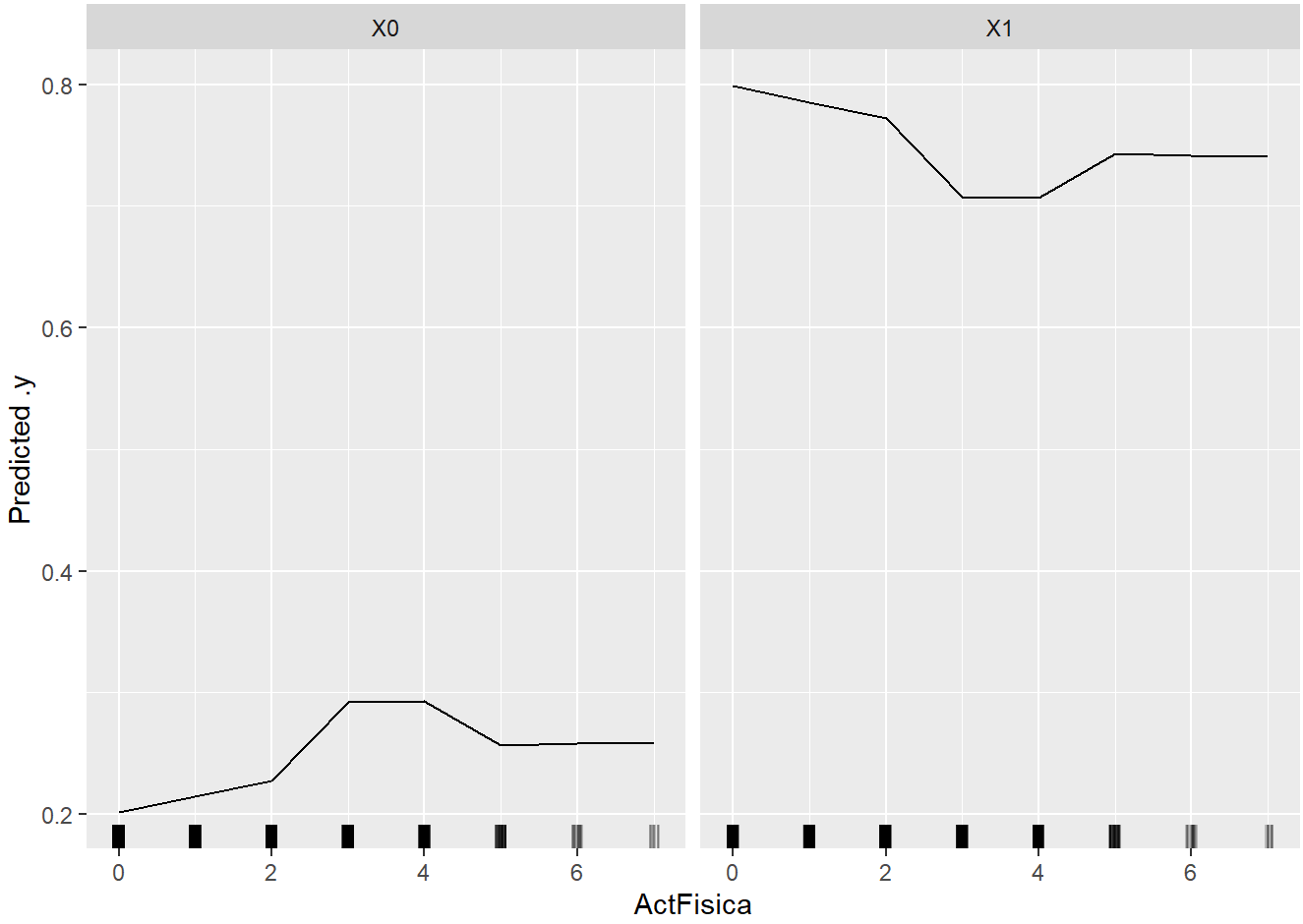
▼ Mostrar el código

```
pdp2 <- FeatureEffect$new(predictor, feature = "Hist_fam", method="pdp")
pdp2$plot()
```



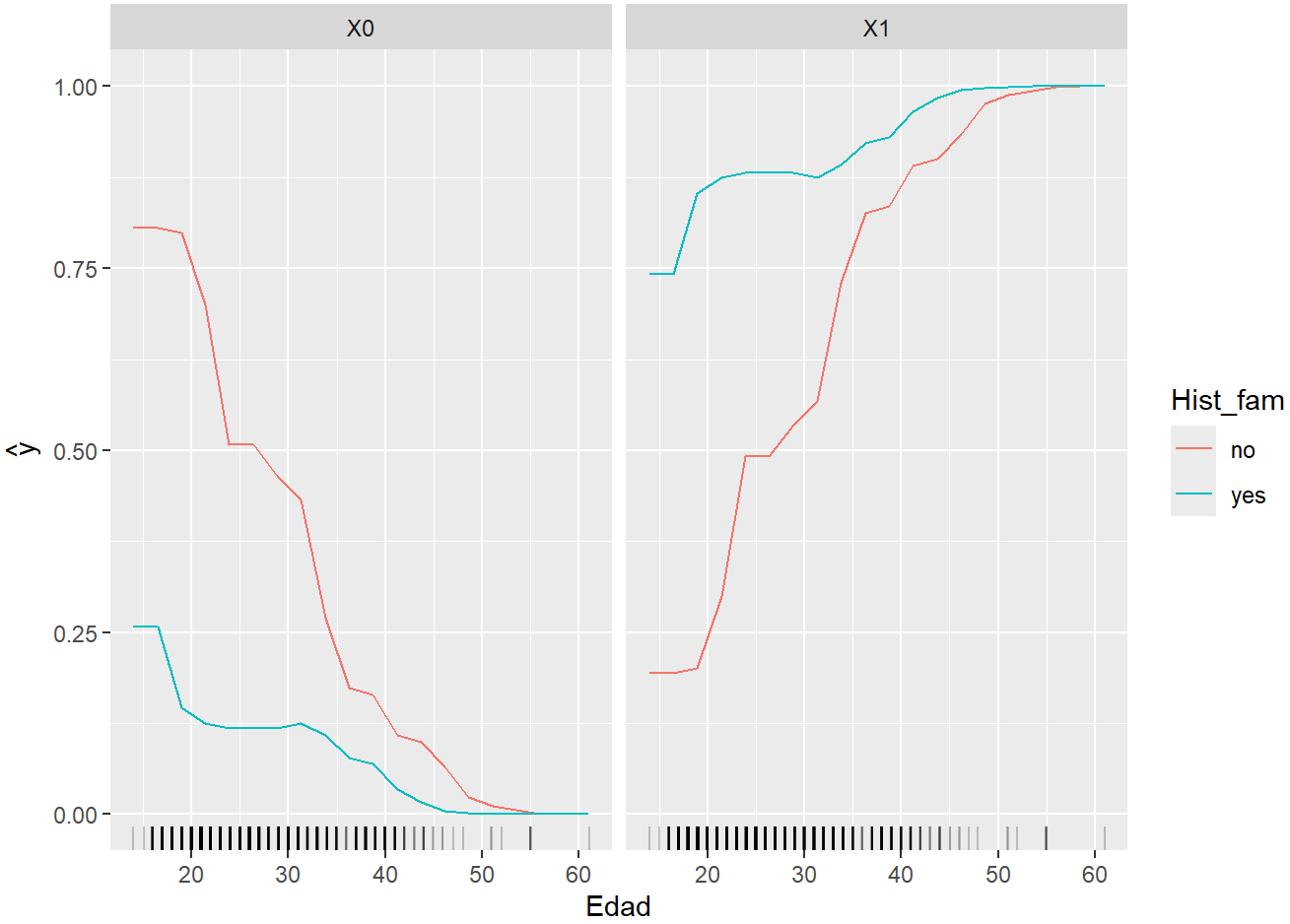
▼ Mostrar el código

```
pdp3 <- FeatureEffect$new(predictor, feature = "ActFisica", method="pdp", grid.size = 8) # Ponemo
pdp3$plot()
```



▼ Mostrar el código

```
pdp4 <- FeatureEffect$new(predictor, feature = c("Hist_fam", "Edad"), method="pdp")
pdp4$plot()
```



Se han construido distintos gráficos utilizando distintas variables. Se muestra cómo se modifican las probabilidades de los eventos (0 y 1) por separado, aunque en este caso sería suficiente con uno de ellos, pues con complementarios. Los gráficos permiten extraer las siguientes conclusiones:

- El efecto de la edad en la probabilidad de tener sobrepeso no es lineal, pero sí es monótono. Se observa como crece has ta que se estanca a partir de los 50 años.
- En líneas generales, las personas que tienen historial familiar de sobrepeso tienen una probabilidad de tener sobrepeso 40 puntos porcentuales superior.
- El efecto de la actividad física resulta contraintuitivo, pues se obtienen probabilidad mayores de sobrepeso entre las personas que más deporte realizan, con respecto a las que lo hacen de manera “moderada”.
- Existe cierta interacción entre el historial familiar y la edad, pues las dos gráficas, aunque similares, tienen un crecimiento distinto. Así mismo, se observa claramente que se trata de dos variables importantes pues existe una diferencia de casi 80 puntos porcentuales en la probabilidad de los extremos.