

# Proyecto

*Jose Antonio Ruiz Millan y Juan Carlos Ruiz Garcia*

*5 de junio de 2018*

## 1. Ajuste del mejor modelo.

Este ejercicio se centra en el ajustar el mejor predictor (lineal o no-lineal) a un conjunto de datos. Debemos mostrar que los distintos algoritmos proponen soluciones para los datos pero que unas soluciones son mejores que otros para unos datos dados. El criterio que usaremos en la comparación será el error medio cuadrático para regresión, la curva ROC en clasificación binaria y el número de errores en clasificación multiclas. Además de un modelo lineal se deberán presentar resultados con al menos dos modelos de entre los propuestos.

Los posibles modelos no-lineales a usar son:

- **Redes Neuronales.** Considerar tres clases de funciones definidas por arquitecturas con 1,2 y 3 capas de unidades ocultas y número de unidades por capa en el rango 0-50. Definir un conjunto de modelos(arquitecturas) y elegir el mejor por validación cruzada. Recordar que a igualdad de  $E_{out}$  siempre es preferible la arquitectura más pequeña.
- **Máquina de Soporte de Vectores (SVM):** usar solo el núcleo RBF-Gaussiano o el polinomial. Encontrar el mejor valor para el parámetro libre hasta una precisión de 2 cifras (enteras o decimales)
- **Boosting:** Para clasificación usar AdaBoost con funciones “stamp”. Para regresión usar árboles como regresores simples.
- **Random Forest:** Usar los valores que por defecto se dan en la teoría y experimentar para obtener el número de árboles adecuado.

Se habrá de buscar el mejor modelo posible para la base de datos seleccionada y se habrá de justificar cada uno de los pasos dados para conseguirlo. Los puntos de discusión señalados en el trabajo.3 deben de servir como guía.

**La base de datos utilizada en nuestro caso es: *Smartphone-Based Recognition of Human Activities and Postural Transitions Data Set***, por lo que nuestro problema es un problema de clasificación.

Antes de nada, comentar que el paquete que vamos a utilizar en todos los casos es el paquete caret, ya que nos permite realizar todas las operaciones con simples instrucciones. Toda la información sobre este paquete se puede ver en <https://topepo.github.io/caret/index.html>

No obstante, los paquetes necesarios para el correcto funcionamiento de la practica son:

- caret
- glmnet
- doParallel
- parallel
- kernlab
- rpart
- adabag
- plyr

## 1. Comprender el problema a resolver.

Los experimentos se llevaron a cabo con un grupo de 30 voluntarios dentro de un grupo de edad de 19-48 años. Realizaron un protocolo de actividades compuesto por seis actividades básicas: tres posturas estáticas (de pie, sentado, acostado) y tres actividades dinámicas (caminar, bajar escaleras y subir escaleras). El experimento también incluyó transiciones posturales que ocurrieron entre las posturas estáticas. Estos son: de pie-a-sentarse, sentarse-a-de pie, sentarse-a-acostarse, acostarse-a-sentarse, de pie-a-acostarse y acostarse-a-de pie. Todos los participantes llevaban un teléfono inteligente (Samsung Galaxy S II) en la cintura durante la ejecución del experimento. Captura la aceleración lineal de 3 ejes y la velocidad angular de 3 ejes a una velocidad constante de 50 Hz utilizando el acelerómetro integrado y el giroscopio del dispositivo. Los experimentos fueron grabados en video para etiquetar los datos manualmente. El conjunto de datos obtenidos se dividió aleatoriamente en dos conjuntos, donde el 70% de los voluntarios se seleccionó para generar los datos de entrenamiento y el 30% de los datos de prueba.

Las señales del sensor (acelerómetro y giroscopio) se preprocesaron mediante la aplicación de filtros de ruido y luego se tomaron muestras en ventanas correderas de ancho fijo de 2,56 segundos y 50% de superposición (128 lecturas / ventana). La señal de aceleración del sensor, que tiene componentes de movimiento gravitacional y corporal, se separó usando un filtro de paso bajo Butterworth en la aceleración del cuerpo y la gravedad. Se supone que la fuerza gravitacional tiene componentes de baja frecuencia, por lo tanto, se utilizó un filtro con una frecuencia de corte de 0.3 Hz. Desde cada ventana, se obtuvo un vector de 561 características calculando variables del dominio de tiempo y frecuencia. Ver 'features\_info.txt' para más detalles.

### Información adicional

- Las características están normalizadas y limitadas dentro de [-1,1].
- Cada vector de características es una fila en los archivos 'X' y 'y'.
- Las unidades utilizadas para las aceleraciones (total y cuerpo) son 'g's (gravedad de la tierra -> 9.80665 m / seg<sup>2</sup>).
- Las unidades del giroscopio son rad / seg.

## 2. Preprocesado los datos: por ejemplo categorización, normalización, etc.

Como indicamos en el apartado anterior, estos datos ya vienen preprocesados, por lo que mostraré con algunas gráficas realmente la distribución de algunos de ellos ya que son demasiadas variables y también realizaremos el preprocesado PCA para eliminar variables y quedarnos con un numero de variables que nos permita explicar un 95% de la varianza de los datos.

```
# Leemos el fichero que vemos a continuación, que contiene
# el nombre de las diferentes actividades que realiza cada
# una de las personas como está indicado en el apartado 1.
temp <- read.table("activity_labels.txt",sep="",header = FALSE)

# Nos quedamos únicamente con estos valores ya que la primera
# columna son identificadores.
activityLabels <- as.character(temp$V2)

#Leemos ahora los datos de train que contiene únicamente los propios datos.
#Leemos también el test para tenerlo ya cargado.
train.x <- read.table("X_train.txt", sep = "",header=FALSE)
test.x <- read.table("X_test.txt", sep = "",header=FALSE)

#Leemos también las etiquetas de estos correspondientes datos.
train.y <- read.table("y_train.txt", sep = "",header=FALSE)
test.y <- read.table("y_test.txt", sep = "",header=FALSE)

#Le asignamos un nombre a la columna, en mi caso será Actividad.
colnames(train.y) <- "Actividad"
```

```

colnames(test.y) <- "Actividad"

#Los pasamos a factor para facilitar su uso posterior.
train.y$Actividad <- as.factor(train.y$Actividad)
test.y$Actividad <- as.factor(test.y$Actividad)

#Enlazamos cada uno de ellos con la actividad a la que pertenece.
levels(train.y$Actividad) <- activityLabels
levels(test.y$Actividad) <- activityLabels

```

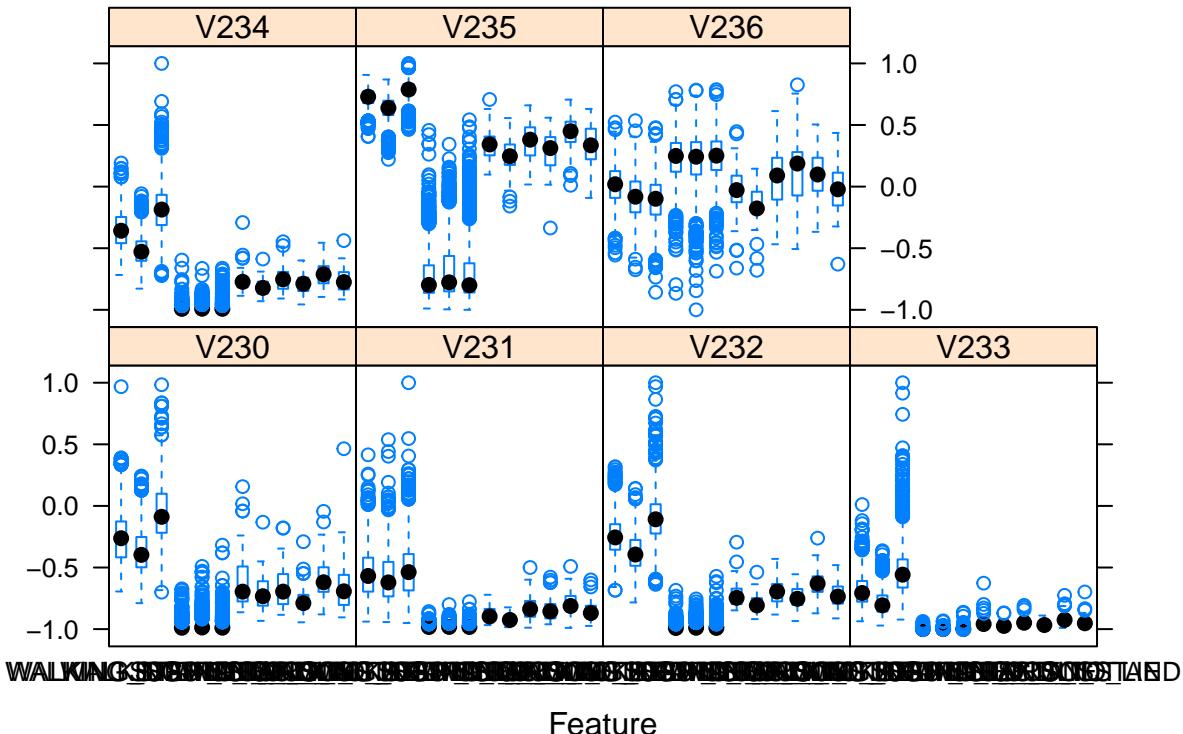
Una vez tenemos los datos vamos a mostrar algunos gráficos para poder visualizar la distribución de los mismos.

```

#Diagrama de caja
featurePlot(x=train.x[,230:236],y=train.y$Actividad,plot="box",
            main = "Diagrama de caja (TRAIN)")

```

**Diagrama de caja (TRAIN)**

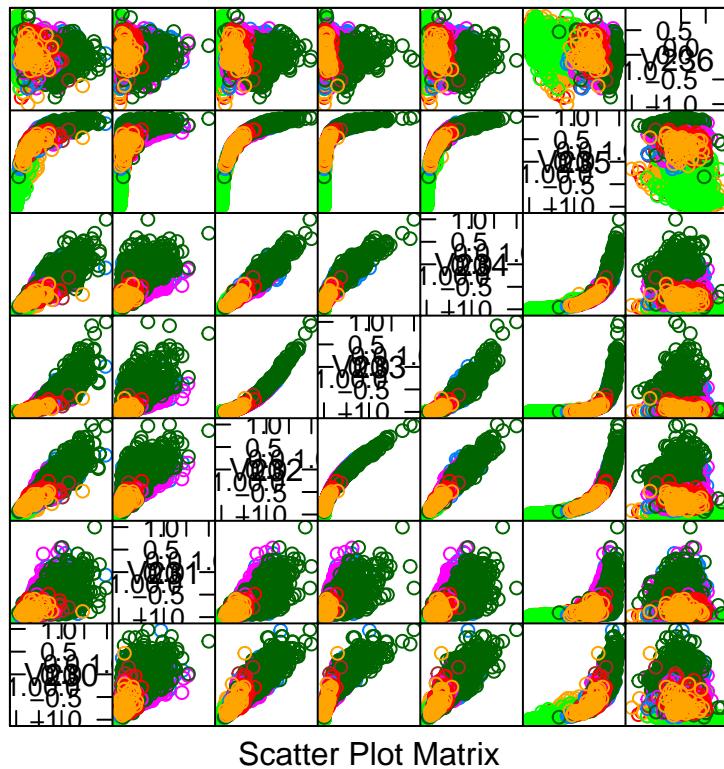


```

#Diagrama de dispersión
featurePlot(x=train.x[,230:236],y=train.y$Actividad,plot="pairs",
            main = "Diagrama de dispersión (TRAIN)")

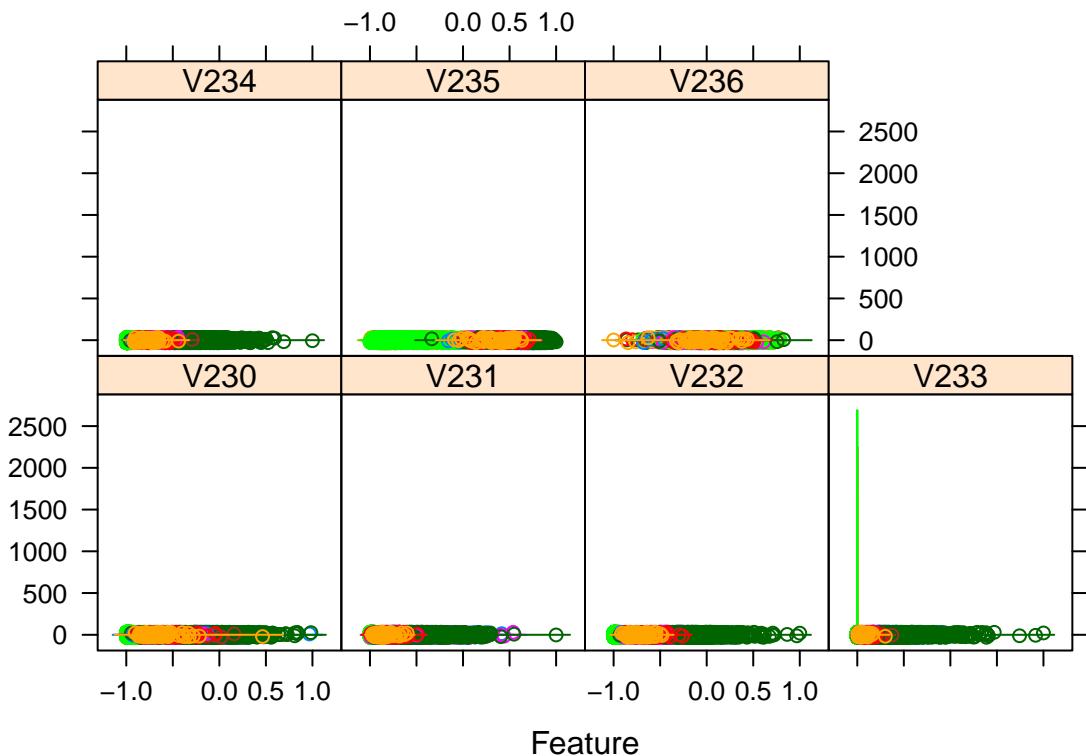
```

## Diagrama de dispersión (TRAIN)



```
#Diagrama de densidad  
featurePlot(x=train.x[,230:236],y=train.y$Actividad,plot="density",  
           main = "Diagrama de densidad (TRAIN)")
```

## Diagrama de densidad (TRAIN)

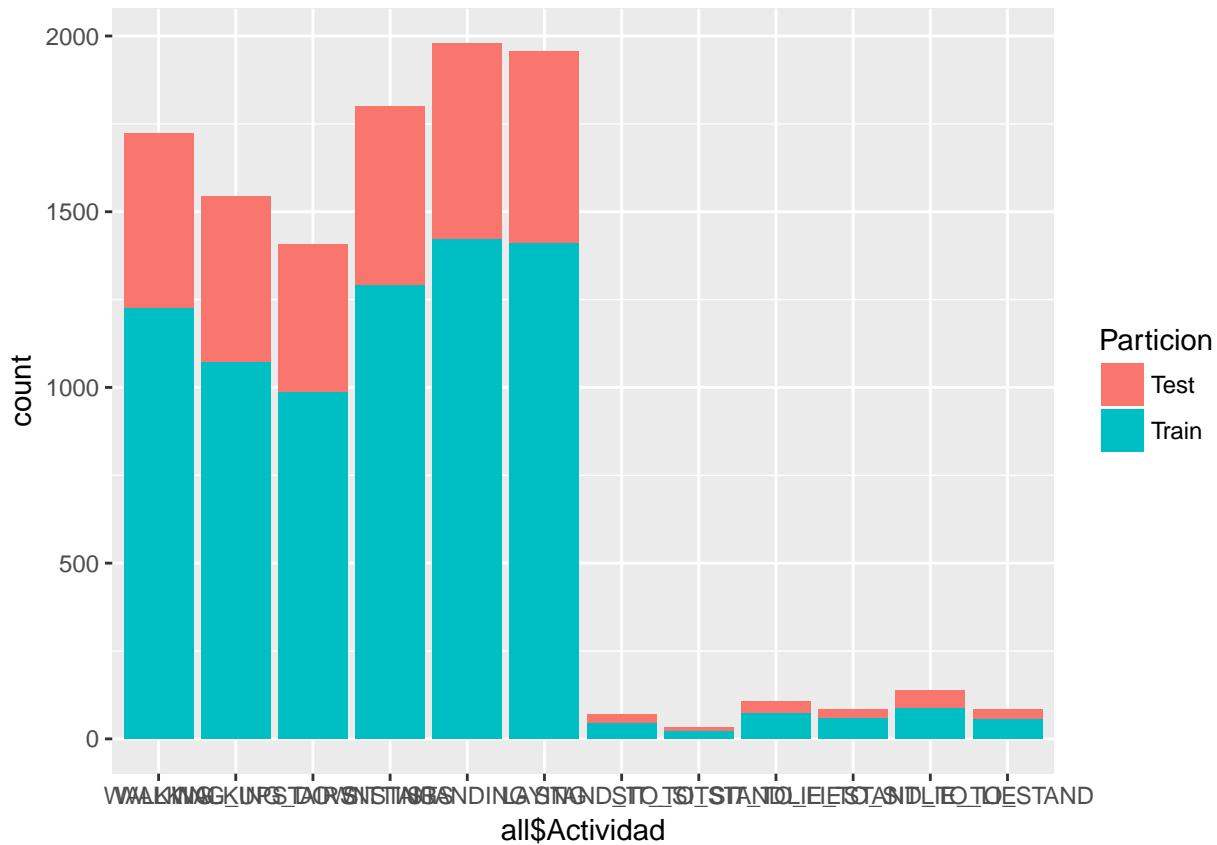


```
#Unimos todos los datos en un sólo conjunto
train <- cbind(train.x,train.y)
test <- cbind(test.x,test.y)

train$Particion = "Train"
test$Particion = "Test"

#Unimos todos los datos, tanto train como test para comprobar su
#distribución en las particiones.
all <- rbind(train, test)

qplot(data=all, x=all$Actividad, fill=Particion)
```



Podemos ver en las gráficas que todos los datos de la misma clase suelen estar compactos y unidos e incluso que entre algunos de ellos existe una buena relación. No obstante vamos a realizar el preprocessado indicado anteriormente para intentar minimizar el espacio. También utilizaremos el preprocessado para eliminar las características que tengan varianza cero.

Podemos visualizar en el último gráfico los datos de train y test están bien distribuidos ya que tenemos más elementos de cada tipo en train que en test y están bien distribuidos.

Además visualizamos que para las últimas 6 clases tenemos muchos menos datos que en el resto, por lo que esto aprenderá al aprendizaje negativamente.

```
#Preprocesamos y mostramos los resultados.
preprocesado <- preprocess(train.x,method = c("nzv","pca"))
print(preprocesado)
```

```
## Created from 7767 samples and 561 variables
##
## Pre-processing:
##   - centered (561)
##   - ignored (0)
##   - principal component signal extraction (561)
##   - scaled (561)
##
## PCA needed 103 components to capture 95 percent of the variance
#Aplicamos los cambios.
train.x.Pre <- predict(preprocesado,train.x)
test.x.Pre <- predict(preprocesado,test.x)
```

Podemos ver como el preprocesado ha sido capaz de dejarnos el número de variables en 103, cuando en un principio teníamos 561. Esto nos dice que hemos conseguido reducir un 81% de las variables para poder explicar el 95% de la varianza.

### 3. Selección de clases de funciones a usar.

En nuestro caso, utilizaremos las clases de funciones lineales para un ajuste lineal utilizando el algoritmo de regresión logística y también utilizaremos modelos no lineales como el SVM y el Boosting. Esto nos permitirá ver qué tipo de modelo y clase se ajusta mejor a estos datos.

### 4. Definición de los conjuntos de training, validación y test usados en su caso.

En este conjunto de datos, tenemos los datos separados, es decir, la definición de los mismos nos dicen que hay un 70% de los datos en train y un 30% de los datos en test. Por lo que esta tarea estaría completada. No obstante, para las particiones de validación usaremos CV (validación cruzada) que los creará automáticamente a partir de los datos del train.

### 5. Discutir la necesidad de regularización y en su caso la función usada para ello.

En nuestro caso, en el modelo lineal vamos a utilizar dos tipos de regularización, L1 regularization (Regularización Lasso) y otro basado en L2 regularization (Regularización Ridge). Nos quedaremos con la que mejor resultado nos ofrezca y ese será el modelo final.

- L1 regularization: Añade una penalización igual a la suma del valor absoluto de los coeficientes.

$$Error_{L1} = Error + \lambda \sum_{i=0}^N |\beta_i|$$

- L2 regularization: Añade una penalización igual a la suma de los coeficientes al cuadrado.

$$Error_{L2} = Error + \lambda \sum_{i=0}^N |\beta_i|^2$$

### 6. Definir los modelos a usar y estimar sus parámetros e hyperparámetros.

## Modelo lineal

Para este apartado, utilizaremos regresión logística multiclas ya que obtuvimos un buen resultado en la práctica anterior y permite solucionar este problema. Utilizaremos dos tipos de regularización para

Para ello, lo primero que vamos a hacer es a través de CV (validación cruzada) obtener los parámetros óptimos y así obtener un buen porcentaje de acierto o al menos intentar obtenerlo.

```
#Funcion para calcular el error
clas.Err <- function(datos,predicciones){
  err <- 0

  for(i in 1:length(predicciones)){
    if(datos[i,ncol(datos)] != predicciones[i]) err <- err+1
  }
  err <- err/length(predicciones)
  err
}

#Definimos el control a utilizar en el que únicamente especificamos que vamos a
#usar validacion cruzada con 10 folds.
ctrl <- trainControl(method="cv", number=10)

#Creamos los dos grid para el entrenamiento. Uno realizara la regularizacion
#Lasso y otro la regularizacion Ridge
GridLasso <- expand.grid(alpha =1,lambda = seq(0.0001,2,length = 100))
GridRidge <- expand.grid(alpha =0,lambda = seq(0.0001,2,length = 100))
```

```

#Paralelizamos el proceso para que se realice más rápido.
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)

#Realizamos los diferentes entrenamientos utilizando glmnet con las distintas regularizaciones.
modelo.lineal.Lasso <- train(x=as.matrix(train.x.Pre) , y=train.y$Actividad ,
                               method ="glmnet", trControl = ctrol,tuneGrid=GridLasso)
modelo.lineal.Ridge <- train(x=as.matrix(train.x.Pre) , y=train.y$Actividad ,
                               method ="glmnet", trControl = ctrol,tuneGrid=GridRidge)

stopCluster(cl)

#Mostramos los hiperparametros de los modelos.
print(modelo.lineal.Lasso)

## glmnet
##
## 7767 samples
## 103 predictors
## 12 classes: 'WALKING', 'WALKING_UPSTAIRS', 'WALKING_DOWNSTAIRS', 'SITTING', 'STANDING', 'LAYING',
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6989, 6990, 6990, 6990, 6992, 6992, ...
## Resampling results across tuning parameters:
##
##     lambda      Accuracy   Kappa
## 0.00010000  0.9609960  0.9538311
## 0.02030101  0.8417667  0.8112317
## 0.04050202  0.7835598  0.7414074
## 0.06070303  0.7206084  0.6655716
## 0.08090404  0.6698802  0.6043306
## 0.10110505  0.6237915  0.5484749
## 0.12130606  0.5086866  0.4081699
## 0.14150707  0.4210115  0.2969654
## 0.16170808  0.3485274  0.2034338
## 0.18190909  0.3408004  0.1935034
## 0.20211010  0.1832110  0.0000000
## 0.22231111  0.1832110  0.0000000
## 0.24251212  0.1832110  0.0000000
## 0.26271313  0.1832110  0.0000000
## 0.28291414  0.1832110  0.0000000
## 0.30311515  0.1832110  0.0000000
## 0.32331616  0.1832110  0.0000000
## 0.34351717  0.1832110  0.0000000
## 0.36371818  0.1832110  0.0000000
## 0.38391919  0.1832110  0.0000000
## 0.40412020  0.1832110  0.0000000
## 0.42432121  0.1832110  0.0000000
## 0.44452222  0.1832110  0.0000000
## 0.46472323  0.1832110  0.0000000
## 0.48492424  0.1832110  0.0000000
## 0.50512525  0.1832110  0.0000000

```

```

## 0.52532626 0.1832110 0.0000000
## 0.54552727 0.1832110 0.0000000
## 0.56572828 0.1832110 0.0000000
## 0.58592929 0.1832110 0.0000000
## 0.60613030 0.1832110 0.0000000
## 0.62633131 0.1832110 0.0000000
## 0.64653232 0.1832110 0.0000000
## 0.66673333 0.1832110 0.0000000
## 0.68693434 0.1832110 0.0000000
## 0.70713535 0.1832110 0.0000000
## 0.72733636 0.1832110 0.0000000
## 0.74753737 0.1832110 0.0000000
## 0.76773838 0.1832110 0.0000000
## 0.78793939 0.1832110 0.0000000
## 0.80814040 0.1832110 0.0000000
## 0.82834141 0.1832110 0.0000000
## 0.84854242 0.1832110 0.0000000
## 0.86874343 0.1832110 0.0000000
## 0.88894444 0.1832110 0.0000000
## 0.90914545 0.1832110 0.0000000
## 0.92934646 0.1832110 0.0000000
## 0.94954747 0.1832110 0.0000000
## 0.96974848 0.1832110 0.0000000
## 0.98994949 0.1832110 0.0000000
## 1.01015051 0.1832110 0.0000000
## 1.03035152 0.1832110 0.0000000
## 1.05055253 0.1832110 0.0000000
## 1.07075354 0.1832110 0.0000000
## 1.09095455 0.1832110 0.0000000
## 1.11115556 0.1832110 0.0000000
## 1.13135657 0.1832110 0.0000000
## 1.15155758 0.1832110 0.0000000
## 1.17175859 0.1832110 0.0000000
## 1.19195960 0.1832110 0.0000000
## 1.21216061 0.1832110 0.0000000
## 1.23236162 0.1832110 0.0000000
## 1.25256263 0.1832110 0.0000000
## 1.27276364 0.1832110 0.0000000
## 1.29296465 0.1832110 0.0000000
## 1.31316566 0.1832110 0.0000000
## 1.33336667 0.1832110 0.0000000
## 1.35356768 0.1832110 0.0000000
## 1.37376869 0.1832110 0.0000000
## 1.39396970 0.1832110 0.0000000
## 1.41417071 0.1832110 0.0000000
## 1.43437172 0.1832110 0.0000000
## 1.45457273 0.1832110 0.0000000
## 1.47477374 0.1832110 0.0000000
## 1.49497475 0.1832110 0.0000000
## 1.51517576 0.1832110 0.0000000
## 1.53537677 0.1832110 0.0000000
## 1.55557778 0.1832110 0.0000000
## 1.57577879 0.1832110 0.0000000
## 1.59597980 0.1832110 0.0000000

```

```

##   1.61618081  0.1832110  0.0000000
##   1.63638182  0.1832110  0.0000000
##   1.65658283  0.1832110  0.0000000
##   1.67678384  0.1832110  0.0000000
##   1.69698485  0.1832110  0.0000000
##   1.71718586  0.1832110  0.0000000
##   1.73738687  0.1832110  0.0000000
##   1.75758788  0.1832110  0.0000000
##   1.777778889 0.1832110  0.0000000
##   1.79798990  0.1832110  0.0000000
##   1.81819091  0.1832110  0.0000000
##   1.83839192  0.1832110  0.0000000
##   1.85859293  0.1832110  0.0000000
##   1.87879394  0.1832110  0.0000000
##   1.89899495  0.1832110  0.0000000
##   1.91919596  0.1832110  0.0000000
##   1.93939697  0.1832110  0.0000000
##   1.95959798  0.1832110  0.0000000
##   1.97979899  0.1832110  0.0000000
##   2.00000000  0.1832110  0.0000000
##
## Tuning parameter 'alpha' was held constant at a value of 1
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were alpha = 1 and lambda = 1e-04.

print(modelo.lineal.Ridge)

## glmnet
##
## 7767 samples
## 103 predictors
## 12 classes: 'WALKING', 'WALKING_UPSTAIRS', 'WALKING_DOWNSTAIRS', 'SITTING', 'STANDING', 'LAYING',
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6992, 6991, 6990, 6993, 6992, 6986, ...
## Resampling results across tuning parameters:
##
##   lambda      Accuracy    Kappa
##   0.00010000  0.9405157  0.9295103
##   0.02030101  0.9400013  0.9288943
##   0.04050202  0.9331777  0.9207362
##   0.06070303  0.9255802  0.9116352
##   0.08090404  0.9196575  0.9044898
##   0.10110505  0.9154059  0.8993792
##   0.12130606  0.9116734  0.8948507
##   0.14150707  0.9087115  0.8912602
##   0.16170808  0.9065235  0.8885980
##   0.18190909  0.9035623  0.8850130
##   0.20211010  0.9018877  0.8829761
##   0.22231111  0.9002124  0.8809514
##   0.24251212  0.8985393  0.8789227
##   0.26271313  0.8973795  0.8775260
##   0.28291414  0.8972511  0.8773679
##   0.30311515  0.8964791  0.8764359

```

```

## 0.32331616 0.8950634 0.8747309
## 0.34351717 0.8944200 0.8739554
## 0.36371818 0.8942913 0.8737967
## 0.38391919 0.8936457 0.8730201
## 0.40412020 0.8932599 0.8725542
## 0.42432121 0.8931296 0.8723950
## 0.44452222 0.8924856 0.8716184
## 0.46472323 0.8917135 0.8706916
## 0.48492424 0.8910695 0.8699190
## 0.50512525 0.8902972 0.8689903
## 0.52532626 0.8904268 0.8691412
## 0.54552727 0.8900397 0.8686729
## 0.56572828 0.8896526 0.8682044
## 0.58592929 0.8896526 0.8682028
## 0.60613030 0.8896525 0.8681989
## 0.62633131 0.8895244 0.8680404
## 0.64653232 0.8887521 0.8671111
## 0.66673333 0.8875946 0.8657219
## 0.68693434 0.8874660 0.8655637
## 0.70713535 0.8874682 0.8655613
## 0.72733636 0.8875972 0.8657090
## 0.74753737 0.8872114 0.8652427
## 0.76773838 0.8872124 0.8652405
## 0.78793939 0.8869547 0.8649310
## 0.80814040 0.8863097 0.8641538
## 0.82834141 0.8861815 0.8639915
## 0.84854242 0.8848929 0.8624430
## 0.86874343 0.8834770 0.8607412
## 0.88894444 0.8824470 0.8595055
## 0.90914545 0.8819312 0.8588809
## 0.92934646 0.8815441 0.8584113
## 0.94954747 0.8811580 0.8579450
## 0.96974848 0.8807709 0.8574764
## 0.98994949 0.8802550 0.8568542
## 1.01015051 0.8790963 0.8554595
## 1.03035152 0.8785819 0.8548373
## 1.05055253 0.8783243 0.8545199
## 1.07075354 0.8769091 0.8528144
## 1.09095455 0.8760067 0.8517285
## 1.11115556 0.8753632 0.8509530
## 1.13135657 0.8739468 0.8492498
## 1.15155758 0.8730466 0.8481646
## 1.17175859 0.8721452 0.8470784
## 1.19195960 0.8709852 0.8456749
## 1.21216061 0.8694391 0.8438111
## 1.23236162 0.8680246 0.8421087
## 1.25256263 0.8664790 0.8402469
## 1.27276364 0.8651907 0.8386940
## 1.29296465 0.8644168 0.8377576
## 1.31316566 0.8628713 0.8358914
## 1.33336667 0.8611990 0.8338780
## 1.35356768 0.8595249 0.8318568
## 1.37376869 0.8584946 0.8306120
## 1.39396970 0.8572091 0.8290611

```

```

##   1.41417071  0.8556628  0.8271910
##   1.43437172  0.8545036  0.8257863
##   1.45457273  0.8533434  0.8243840
##   1.47477374  0.8511557  0.8217414
##   1.49497475  0.8503808  0.8207983
##   1.51517576  0.8490940  0.8192387
##   1.53537677  0.8472907  0.8170568
##   1.55557778  0.8456167  0.8150300
##   1.57577879  0.8448437  0.8140914
##   1.59597980  0.8434299  0.8123790
##   1.61618081  0.8413735  0.8098905
##   1.63638182  0.8389271  0.8069303
##   1.65658283  0.8359666  0.8033459
##   1.67678384  0.8330060  0.7997649
##   1.69698485  0.8308169  0.7971161
##   1.71718586  0.8279819  0.7936865
##   1.73738687  0.8254052  0.7905659
##   1.75758788  0.8228297  0.7874427
##   1.77778889  0.8196174  0.7835528
##   1.79798990  0.8165278  0.7798017
##   1.81819091  0.8149811  0.7779163
##   1.83839192  0.8115023  0.7736908
##   1.85859293  0.8084102  0.7699288
##   1.87879394  0.8067353  0.7678897
##   1.89899495  0.8054476  0.7663252
##   1.91919596  0.8031275  0.7634981
##   1.93939697  0.8015842  0.7616197
##   1.95959798  0.7988800  0.7583299
##   1.97979899  0.7960465  0.7548749
##   2.00000000  0.7919249  0.7498647
##
## Tuning parameter 'alpha' was held constant at a value of 0
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were alpha = 0 and lambda = 1e-04.

#Calculamos el E_in de los distintos modelo que hemos creado.
predicciones.lineal.Lasso <- predict(modelo.lineal.Lasso,train.x.Pre)
predicciones.lineal.Ridge <- predict(modelo.lineal.Ridge,train.x.Pre)

#Mostramos los distintos errores que hemos obtenido
print(clas.Err(cbind(train.x.Pre,train.y),predicciones.lineal.Lasso))

## [1] 0.01956998
print(clas.Err(cbind(train.x.Pre,train.y),predicciones.lineal.Ridge))

## [1] 0.04712244

```

Hemos obtenido un resultado mejor en la regularización Lasso que en la regularización Ridge, por lo que por ahora, seleccionaremos de estos dos, el que utiliza regularización Lasso.

## SVM

```

#Creamos el control
ctrl <- trainControl(method="cv", number=10)

```

```

#Creamos para la paralelizacion
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)

#Ejecutamos los train para crear los modelos.
modelo.no_lineal.SVM.RBF <- train(x=as.matrix(train.x.Pre) , y=train.y$Actividad,
                                    method ="svmRadial", trControl = ctrl)
modelo.no_lineal.SVM.Polynomial <- train(x=as.matrix(train.x.Pre) , y=train.y$Actividad,
                                            method ="svmPoly", trControl = ctrl)

#Cerramos la paralelizacion
stopCluster(cl)

#Mostramos los hiperparametros de los modelos
print(modelo.no_lineal.SVM.RBF)

## Support Vector Machines with Radial Basis Function Kernel
##
## 7767 samples
## 103 predictors
## 12 classes: 'WALKING', 'WALKING_UPSTAIRS', 'WALKING_DOWNSTAIRS', 'SITTING', 'STANDING', 'LAYING',
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6990, 6992, 6989, 6993, 6991, 6989, ...
## Resampling results across tuning parameters:
##
##     C      Accuracy   Kappa
##     0.25  0.9249449  0.9108146
##     0.50  0.9500521  0.9408145
##     1.00  0.9612510  0.9541113
##
## Tuning parameter 'sigma' was held constant at a value of 0.005977268
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.005977268 and C = 1.

print(modelo.no_lineal.SVM.Polynomial)

## Support Vector Machines with Polynomial Kernel
##
## 7767 samples
## 103 predictors
## 12 classes: 'WALKING', 'WALKING_UPSTAIRS', 'WALKING_DOWNSTAIRS', 'SITTING', 'STANDING', 'LAYING',
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6991, 6989, 6988, 6991, 6990, 6992, ...
## Resampling results across tuning parameters:
##
##     degree  scale   C      Accuracy   Kappa
##     1        0.001  0.25  0.8942945  0.8738623
##     1        0.001  0.50  0.9183756  0.9030415
##     1        0.001  1.00  0.9366544  0.9249347
##     1        0.010  0.25  0.9466934  0.9368676

```

```

##   1      0.010  0.50  0.9515897  0.9426768
##   1      0.010  1.00  0.9548065  0.9464982
##   1      0.100  0.25  0.9558346  0.9477202
##   1      0.100  0.50  0.9571215  0.9492431
##   1      0.100  1.00  0.9594398  0.9519916
##   2      0.001  0.25  0.9212103  0.9064317
##   2      0.001  0.50  0.9380716  0.9266179
##   2      0.001  1.00  0.9469523  0.9371651
##   2      0.010  0.25  0.9615033  0.9544208
##   2      0.010  0.50  0.9649787  0.9585374
##   2      0.010  1.00  0.9671711  0.9611377
##   2      0.100  0.25  0.9640821  0.9574743
##   2      0.100  0.50  0.9642106  0.9576267
##   2      0.100  1.00  0.9642106  0.9576267
##   3      0.001  0.25  0.9336945  0.9213848
##   3      0.001  0.50  0.9448954  0.9347164
##   3      0.001  1.00  0.9517177  0.9428210
##   3      0.010  0.25  0.9653666  0.9589950
##   3      0.010  0.50  0.9671717  0.9611343
##   3      0.010  1.00  0.9683322  0.9625101
##   3      0.100  0.25  0.9617636  0.9546785
##   3      0.100  0.50  0.9617636  0.9546785
##   3      0.100  1.00  0.9617636  0.9546785
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were degree = 3, scale = 0.01 and C
## = 1.

```

```

#Calculamos el E_in de los distintos modelo que hemos creado.
predicciones.no_lineal.SVM.RBF <- predict(modelo.no_lineal.SVM.RBF,train.x.Pre)
predicciones.no_lineal.SVM.Polynomial <- predict(modelo.no_lineal.SVM.Polynomial,
                                                   train.x.Pre)

```

```

#Mostramos los distintos errores que hemos obtenido
print(clas.Err(cbind(train.x.Pre,train.y),predicciones.no_lineal.SVM.RBF))

```

```

## [1] 0.01673748
print(clas.Err(cbind(train.x.Pre,train.y),predicciones.no_lineal.SVM.Polynomial))

```

```

## [1] 0.001931248

```

Podemos ver como obtenemos un error menor en el polinomial que en el RBF e incluso que en la regresión logísitca por lo que por el momento, el modelo a utilizar será SVM Polinomial. Los mejores valores obtenidos han sido  $degree=3$ ,  $scale=0.01$  y  $C=1$ .

## Boosting

```

#Creamos el control
ctrol <- trainControl(method="cv", number=10)

#Paralelizamos el proceso para que se realice más rapido.
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)

```

```

#Creamos el modelo
modelo.no_lineal.boosting <- train(x=as.matrix(train.x.Pre) , y=train.y$Actividad ,
                                     method ="AdaBoost.M1", trControl = ctrl)

#Terminamos la paralelizacion
stopCluster(cl)

#Mostramos los hiperparametros del modelo
print(modelo.no_lineal.boosting)

## AdaBoost.M1
##
## 7767 samples
## 103 predictors
##   12 classes: 'WALKING', 'WALKING_UPSTAIRS', 'WALKING_DOWNSTAIRS', 'SITTING', 'STANDING', 'LAYING',
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6990, 6991, 6992, 6989, 6991, 6989, ...
## Resampling results across tuning parameters:
##
##   coeflearn  maxdepth  mfinal  Accuracy  Kappa
##   Breiman    1         50      0.3402857  0.2039100
##   Breiman    1         100     0.3404144  0.2040649
##   Breiman    1         150     0.3404144  0.2040649
##   Breiman    2         50      0.6338419  0.5613007
##   Breiman    2         100     0.6467099  0.5770976
##   Breiman    2         150     0.6259568  0.5516613
##   Breiman    3         50      0.7525446  0.7063831
##   Breiman    3         100     0.7564049  0.7109289
##   Breiman    3         150     0.7556319  0.7099944
##   Freund     1         50      0.3402859  0.2039107
##   Freund     1         100     0.3402859  0.2039100
##   Freund     1         150     0.3404144  0.2040649
##   Freund     2         50      0.6541800  0.5862912
##   Freund     2         100     0.6518618  0.5834586
##   Freund     2         150     0.6519905  0.5836193
##   Freund     3         50      0.7453357  0.6979934
##   Freund     3         100     0.7463600  0.6991110
##   Freund     3         150     0.7479036  0.7008918
##   Zhu        1         50      0.5657306  0.4872408
##   Zhu        1         100     0.6100342  0.5395833
##   Zhu        1         150     0.6140044  0.5445787
##   Zhu        2         50      0.7036147  0.6496749
##   Zhu        2         100     0.7249919  0.6750245
##   Zhu        2         150     0.7265429  0.6768597
##   Zhu        3         50      0.7475065  0.7015472
##   Zhu        3         100     0.7834388  0.7439784
##   Zhu        3         150     0.8063493  0.7710119
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mfinal = 150, maxdepth = 3
## and coeflearn = Zhu.

```

```

#Obtenemos las predicciones
prediccciones.no_lineal.boosting <- predict(modelo.no_lineal.boosting,train.x.Pre)

#Mostramos el error
clas.Err(cbind(train.x.Pre,train.y),prediccciones.no_lineal.boosting)

## [1] 0.1856573

```

Vemos que este modelo es el peor de todos, por lo que de los 5 modelos usados (2 de regresión logistica lineal, 2 de SVM y boosting) nos quedaremos como modelo final el **SVM Polinomial**, ya que es el que nos ofrece un  $E_{in}$  mas bajo.

## 7. Selección y ajuste modelo final.

Como hemos dicho anteriormente, ya tenemos elegido el modelo final, por lo que en este paso lo queharemos será entrenar un nuevo modelo utlizando el mismo algoritmo (SVM Polinomial), pero en este caso utilizando la totalidad de los datos del train y los parámetros optimos que obtuvimos con la validación cruzada.

```

#Especificamos los parámetros óptimos
GridSVMpoly <- expand.grid(scale = modelo.no_lineal.SVM.Polynomial[["bestTune"]][["scale"]],
                             degree = modelo.no_lineal.SVM.Polynomial[["bestTune"]][["degree"]],
                             C = modelo.no_lineal.SVM.Polynomial[["bestTune"]][["C"]])

#Paralelizamos el proceso para que se realice más rapido.
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)

#Creamos el modelo
modelo.final <- train(x=as.matrix(train.x.Pre) , y=train.y$Actividad ,
                        method ="svmPoly", tuneGrid = GridSVMpoly)

#Terminamos la paralelizacion
stopCluster(cl)

# Mostramos el modelo final
print(modelo.final)

## Support Vector Machines with Polynomial Kernel
##
## 7767 samples
## 103 predictors
## 12 classes: 'WALKING', 'WALKING_UPSTAIRS', 'WALKING_DOWNSTAIRS', 'SITTING', 'STANDING', 'LAYING',
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 7767, 7767, 7767, 7767, 7767, 7767, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9610763  0.9538911
##
## Tuning parameter 'degree' was held constant at a value of 3
##
## Tuning parameter 'scale' was held constant at a value of 0.01
##
## Tuning parameter 'C' was held constant at a value of 1

```

```
#Obtenemos las predicciones
predicciones.final <- predict(modelo.final,train.x.Pre)
```

```
#Mostramos el error
clas.Err(cbind(train.x.Pre,train.y),predicciones.final)
```

```
## [1] 0.001931248
```

## 8. Discutir la idoneidad de la métrica usada en el ajuste

Para ello, vamos a utilizar la matriz de confusión que nos permite ver perfectamente como el predictor está realizando su trabajo y tambien podemos ver dónde específicamente se está equivocando el clasificador.

```
confusionMatrix(data=train.y$Actividad, reference = as.factor(predicciones.final))
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction      WALKING WALKING_UPSTAIRS WALKING_DOWNSTAIRS SITTING
##   WALKING          1226            0                  0           0
##   WALKING_UPSTAIRS       0         1073                  0           0
##   WALKING_DOWNSTAIRS      0            0                  987           0
##   SITTING            0            0                  0           1285
##   STANDING            0            0                  0            7
##   LAYING              0            0                  0           0
##   STAND_TO_SIT          0            0                  0           0
##   SIT_TO_STAND          0            0                  0           0
##   SIT_TO_LIE            0            0                  0           0
##   LIE_TO_SIT            0            0                  0           0
##   STAND_TO_LIE          0            0                  0           0
##   LIE_TO_STAND          0            0                  0           0
##
##          Reference
## Prediction      STANDING LAYING STAND_TO_SIT SIT_TO_STAND SIT_TO_LIE
##   WALKING            0     0        0             0           0
##   WALKING_UPSTAIRS      0     0        0             0           0
##   WALKING_DOWNSTAIRS      0     0        0             0           0
##   SITTING              8     0        0             0           0
##   STANDING          1416     0        0             0           0
##   LAYING              0    1413       0             0           0
##   STAND_TO_SIT          0     0        47            0           0
##   SIT_TO_STAND          0     0        0            23           0
##   SIT_TO_LIE            0     0        0             0           75
##   LIE_TO_SIT            0     0        0             0           0
##   STAND_TO_LIE          0     0        0             0           0
##   LIE_TO_STAND          0     0        0             0           0
##
##          Reference
## Prediction      LIE_TO_SIT STAND_TO_LIE LIE_TO_STAND
##   WALKING            0     0            0
##   WALKING_UPSTAIRS      0     0            0
##   WALKING_DOWNSTAIRS      0     0            0
##   SITTING              0     0            0
##   STANDING            0     0            0
##   LAYING              0     0            0
##   STAND_TO_SIT          0     0            0
##   SIT_TO_STAND          0     0            0
```

```

##   SIT_TO_LIE          0          0          0
##   LIE_TO_SIT          60         0          0
##   STAND_TO_LIE         0         90          0
##   LIE_TO_STAND         0         0          57
##
## Overall Statistics
##
##           Accuracy : 0.9981
## 95% CI : (0.9968, 0.9989)
## No Information Rate : 0.1833
## P-Value [Acc > NIR] : < 2.2e-16
##
##           Kappa : 0.9977
## McNemar's Test P-Value : NA
##
## Statistics by Class:
##
##           Class: WALKING Class: WALKING_UPSTAIRS
## Sensitivity          1.0000          1.0000
## Specificity          1.0000          1.0000
## Pos Pred Value       1.0000          1.0000
## Neg Pred Value       1.0000          1.0000
## Prevalence           0.1578          0.1381
## Detection Rate       0.1578          0.1381
## Detection Prevalence 0.1578          0.1381
## Balanced Accuracy    1.0000          1.0000
##
##           Class: WALKING_DOWNSTAIRS Class: SITTING
## Sensitivity          1.0000          0.9946
## Specificity          1.0000          0.9988
## Pos Pred Value       1.0000          0.9938
## Neg Pred Value       1.0000          0.9989
## Prevalence           0.1271          0.1663
## Detection Rate       0.1271          0.1654
## Detection Prevalence 0.1271          0.1665
## Balanced Accuracy    1.0000          0.9967
##
##           Class: STANDING Class: LAYING Class: STAND_TO_SIT
## Sensitivity          0.9944          1.0000          1.000000
## Specificity          0.9989          1.0000          1.000000
## Pos Pred Value       0.9951          1.0000          1.000000
## Neg Pred Value       0.9987          1.0000          1.000000
## Prevalence           0.1833          0.1819          0.006051
## Detection Rate       0.1823          0.1819          0.006051
## Detection Prevalence 0.1832          0.1819          0.006051
## Balanced Accuracy    0.9966          1.0000          1.000000
##
##           Class: SIT_TO_STAND Class: SIT_TO_LIE
## Sensitivity          1.000000        1.000000
## Specificity          1.000000        1.000000
## Pos Pred Value       1.000000        1.000000
## Neg Pred Value       1.000000        1.000000
## Prevalence           0.002961        0.009656
## Detection Rate       0.002961        0.009656
## Detection Prevalence 0.002961        0.009656
## Balanced Accuracy    1.000000        1.000000
##
##           Class: LIE_TO_SIT Class: STAND_TO_LIE

```

```

## Sensitivity           1.000000 1.000000
## Specificity          1.000000 1.000000
## Pos Pred Value       1.000000 1.000000
## Neg Pred Value       1.000000 1.000000
## Prevalence            0.007725 0.01159
## Detection Rate        0.007725 0.01159
## Detection Prevalence 0.007725 0.01159
## Balanced Accuracy     1.000000 1.000000
##                               Class: LIE_TO_STAND
## Sensitivity           1.000000
## Specificity          1.000000
## Pos Pred Value       1.000000
## Neg Pred Value       1.000000
## Prevalence            0.007339
## Detection Rate        0.007339
## Detection Prevalence 0.007339
## Balanced Accuracy     1.000000

```

Nos damos cuenta, que el modelo utilizado nos da una predicción muy buena. Teniendo una media de acierto del 99% y pudiendo afirmar que a un 95% de confianza el porcentaje de acierto estará entre (0.9968, 0.9989), con un valor p-value < 2.2e-16.

Por otro lado, nos fijamos que nuestro modelo únicamente comete errores en las acciones de SITTING (estar sentado) y STANDING (estar de pie), cosa que es bastante lógica, ya que son posiciones muy similares respecto a los datos tomados de los usuarios que han realizado el experimento y aunque el error es mínimo no es extraño que se cometan.

## 9. Estimacion del error Eout del modelo lo más ajustada posible.

Ahora calcularemos las predicciones con el modelo creado anteriormente y por último estimaremos el  $E_{out}$  con la función definida en apartados anteriores que nos devuelve el porcentaje de error en clasificación de los datos totales.

```

#Predecimos las nuevas etiquetas
predicciones.test <- predict(modelo.final, test.x.Pre)

#Mostramos el error
clas.Err(cbind(test.x.Pre,test.y),predicciones.test)

## [1] 0.08444023

#Mostramos la matriz de confusión para visualizar mejor los errores
#y los aciertos obtenidos
confusionMatrix(data=test.y$Actividad, reference = as.factor(predicciones.test))

## Confusion Matrix and Statistics
##
##                                     Reference
## Prediction
## WALKING          WALKING WALKING_UPSTAIRS WALKING_DOWNSTAIRS SITTING
## WALKING          491           2                 3             0
## WALKING_UPSTAIRS 52            399                20            0
## WALKING_DOWNSTAIRS 7            31                 382            0
## SITTING          0             1                 0             441
## STANDING         0             0                 0             37
## LAYING           0             0                 0             0
## STAND_TO_SIT     0             3                 0             1
## SIT_TO_STAND     0             0                 0             0
## SIT_TO_LIE        0             2                 0             0

```

```

##    LIE_TO_SIT          0          0          0          1
##    STAND_TO_LIE         1          1          0          0
##    LIE_TO_STAND         0          0          0          0
##
##                               Reference
## Prediction      STANDING LAYING STAND_TO_SIT SIT_TO_STAND SIT_TO_LIE
## WALKING           0       0       0       0       0
## WALKING_UPSTAIRS 0       0       0       0       0
## WALKING_DOWNSTAIRS 0       0       0       0       0
## SITTING           66      0       0       0       0
## STANDING          518     0       1       0       0
## LAYING             0      544      1       0       0
## STAND_TO_SIT      2       0      16      0       0
## SIT_TO_STAND      0       0       0      10      0
## SIT_TO_LIE         0       0       0       0      20
## LIE_TO_SIT         0       0       0       0       0
## STAND_TO_LIE       1       1       0       0       7
## LIE_TO_STAND       1       0       2       0       0
##
##                               Reference
## Prediction      LIE_TO_SIT STAND_TO_LIE LIE_TO_STAND
## WALKING           0       0       0
## WALKING_UPSTAIRS 0       0       0
## WALKING_DOWNSTAIRS 0       0       0
## SITTING            0       0       0
## STANDING           0       0       0
## LAYING              0       0       0
## STAND_TO_SIT       0       1       0
## SIT_TO_STAND       0       0       0
## SIT_TO_LIE          0      10      0
## LIE_TO_SIT          20      0       4
## STAND_TO_LIE        0      38      0
## LIE_TO_STAND        7       1      16
##
## Overall Statistics
##
##                         Accuracy : 0.9156
##                         95% CI : (0.9053, 0.925)
## No Information Rate : 0.186
## P-Value [Acc > NIR] : < 2.2e-16
##
##                         Kappa : 0.9004
## Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                         Class: WALKING Class: WALKING_UPSTAIRS
## Sensitivity            0.8911            0.9089
## Specificity            0.9981            0.9736
## Pos Pred Value         0.9899            0.8471
## Neg Pred Value         0.9775            0.9851
## Prevalence              0.1743            0.1388
## Detection Rate          0.1553            0.1262
## Detection Prevalence    0.1569            0.1490
## Balanced Accuracy       0.9446            0.9412
##
##                         Class: WALKING_DOWNSTAIRS Class: SITTING

```

```

## Sensitivity          0.9432      0.9187
## Specificity         0.9862      0.9750
## Pos Pred Value     0.9095      0.8681
## Neg Pred Value     0.9916      0.9853
## Prevalence          0.1281      0.1518
## Detection Rate     0.1208      0.1395
## Detection Prevalence 0.1328      0.1607
## Balanced Accuracy   0.9647      0.9469
##                         Class: STANDING Class: LAYING Class: STAND_TO_SIT
## Sensitivity          0.8810      0.9982      0.800000
## Specificity          0.9852      0.9996      0.997772
## Pos Pred Value       0.9317      0.9982      0.695652
## Neg Pred Value       0.9731      0.9996      0.998726
## Prevalence           0.1860      0.1724      0.006325
## Detection Rate       0.1638      0.1720      0.005060
## Detection Prevalence 0.1758      0.1724      0.007274
## Balanced Accuracy    0.9331      0.9989      0.898886
##                         Class: SIT_TO_STAND Class: SIT_TO_LIE
## Sensitivity          1.000000   0.740741
## Specificity          1.000000   0.996172
## Pos Pred Value       1.000000   0.625000
## Neg Pred Value       1.000000   0.997764
## Prevalence           0.003163   0.008539
## Detection Rate       0.003163   0.006325
## Detection Prevalence 0.003163   0.010120
## Balanced Accuracy    1.000000   0.868456
##                         Class: LIE_TO_SIT Class: STAND_TO_LIE
## Sensitivity          0.740741   0.760000
## Specificity          0.998405   0.99647
## Pos Pred Value       0.800000   0.77551
## Neg Pred Value       0.997769   0.99615
## Prevalence           0.008539   0.01581
## Detection Rate       0.006325   0.01202
## Detection Prevalence 0.007906   0.01550
## Balanced Accuracy    0.869573   0.87823
##                         Class: LIE_TO_STAND
## Sensitivity          0.800000
## Specificity          0.996499
## Pos Pred Value       0.592593
## Neg Pred Value       0.998724
## Prevalence           0.006325
## Detection Rate       0.005060
## Detection Prevalence 0.008539
## Balanced Accuracy    0.898250

```

Vemos que como hemos comentando en apartados anteriores, al tener menos datos en algunas clases, hemos obtenido más fallos sobre dichas clases. Esto es totalmente normal, ya que el aprendizaje sobre estas ha sido mucho menor que sobre el resto de clases. No obstante, han aumentado los errores en algunas de las clases, aunque el principal fallo sigue siendo entre la clase SITTING (estar sentado) y STANDING (estar de pie) al igual que en el aprendizaje.

Aún así, el porcentaje de acierto obtenido en el test es del 91%, valor más que aceptable.

**10. Discutir y justificar la calidad del modelo encontrado y las razones por las que considera que dicho modelo es un buen ajuste que representa adecuadamente los datos muestrales.**

Durante el desarrollo de los experimentos realizados con los distintos modelos utilizados, hemos obtenido:

- En regresión logística Lasso: un error en los datos del train de 0.019 y en el entrenamiento con CV (Cross-Validation) un error de 0.04.
- En regresión logística Ridge: un error en los datos del train de 0.04 y en el entrenamiento con CV (Cross-Validation) un error de 0.05.
- En SVM-RBF: un error en los datos del train de 0.016 y en el entrenamiento con CV (Cross-Validation) un error de 0.04.
- En SVM-Polinomy: un error en los datos del train de 0.0019 y en el entrenamiento con CV (Cross-Validation) un error de 0.031.
- En Boosting: un error en los datos del train de 0.163 y en el entrenamiento con CV (Cross-Validation) un error de 0.198.

Basandonos en estos resultados, decidimos seleccionar como modelo final el **SVM-Polinomy** ya que es el que mejor resultados nos daba. Hemos comprobado utilizando como métrica la **matriz de confusión** que realmente era un modelo de calidad.

Por otro lado, utilizando este modelo, hemos obtenidos un error de 0.084 en el test, lo cual es un resultado muy bueno y por tanto podemos constatar que el modelo elegido/seleccionado es un modelo factible para este problema concreto.