Minería de datos: Preprocesamiento y Clasificación

Competición Kaggle

Máster en Ciencia de Datos e Ingeniería de los Computadores

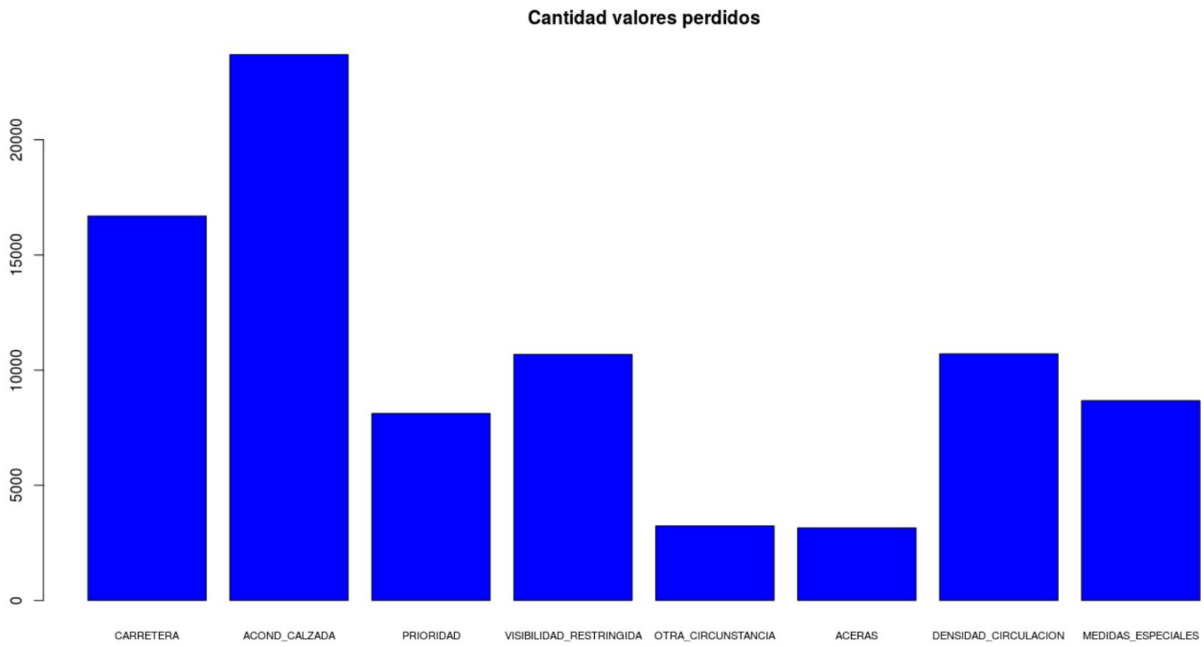
Besay Montesdeoca Santana Graduado en Ingeniería Informática

Nombre de usuario de Kaggle: BesayMontesdeocaSantana

1. Descripción de la base de datos

La base de datos con la que trabajaremos en esta competición se llama *Accidentes* y recoge aproximadamente unos 30000 casos de accidentes de tráfico ocurridos en españa entre los años 2008 y 2013. Para cada accidente disponemos de 29 atributos o variables que describen dicho accidente y que recopilan datos como el mes, la hora, la provincia, el número de vehículos implicados o el tipo de via del lugar donde aconteció el accidente entre otros. La base de datos posee de 6 clases o tipos de accidentes distintos(Atropello, Colisión_Obstáculo, Colisión_Vehículos, Otro, Salida_Vía y Vuelco).

Un gran inconveniente que tenemos con nuestra base de datos es la existencia de valores perdidos en algunas de las variables por lo que a continuación se aplicarán una serie de técnicas de preprocesamiento que intentarán resolver este problema entre otros:



2. Técnicas de preprocesamiento

En un primer vistazo de los valores de las variables de la base de datos se observó que la variable hora poseía demasiados labels, en torno a 448, por lo que se procedió a realizar una transformación de la variable y por tanto a construir los intervalos para simplificar la información de la misma.

```
# Transformamos la variable HORA y creamos intervalos{dat|tst}
transHora = function(dataBase){
 h = as.character(dataBase[["HORA"]])
 h = gsub(",", ".", h)
 h = as.numeric(h)
 dataBase[["HORA"]] = ordered(cut(h, c(-1, 6, 14, 20, 24)), labels = c("madrugada", "mañana", "tarde",
"noche"))
 return(dataBase)
}
dataTrain = transHora(dataTrain)
dataTest = transHora(dataTest)
```

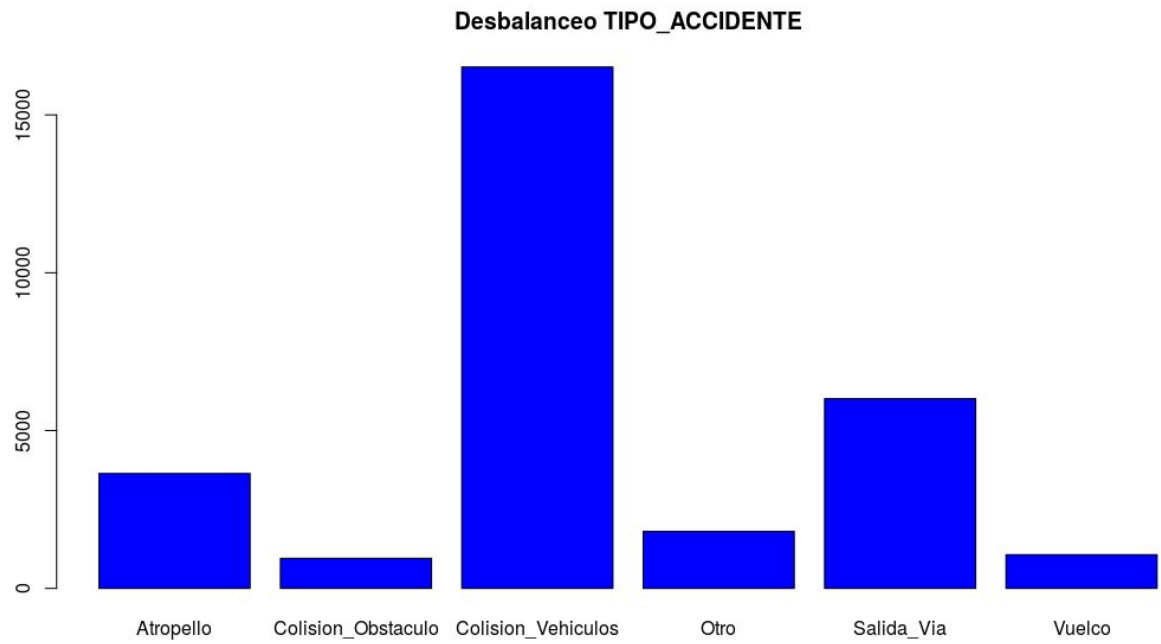
Seguidamente se procedió a paliar el problema de los valores perdidos, para ello nos apoyamos en el paquete *mice* para realizar un proceso de imputación de cada una de las variables con valores perdidos, pero antes se eliminaron algunas variables de la base de datos que, debido posiblemente a su gran variabilidad, dieron problemas a la hora de realizar el proceso de imputación.

Se hizo un primer intento de imputación de la base de datos con todas las variables que poseían valores perdidos, pero el proceso se estancó y no terminó de ejecutar. Esto puede ser debido a que como las funciones de imputación del paquete *MICE* se basan en las demás variables de la base de datos para calcular un valor sintético que sustituya cada uno de los valores perdidos de cada una de las variables, no es viable con el gran número de variables tanto a imputar como variables de referencia en esta base de datos. Por esta razón se procedió a imputar las variables una a una y escribirlas en un fichero para luego ser sustituidas y usadas si es necesario en la clasificación. En este proceso se eliminaron algunas variables que por su distribución impedían la ejecución del algoritmo de imputación.

```
detectNA = function(dataBase){
 p = sapply(dataBase, function(x) sum(is.na(x)))
 p[p > 0]
 nNA = sapply(names(p[p > 0]), function(x){return(which(colnames(dataBase) == x))})
 notNA = sapply(names(p[p == 0]), function(x){return(which(colnames(dataBase) == x))})
   names(p == 0)
 return(list(notNA,"", nNA))
}

imputationMice = function(dataBase, nameVar){
 init = mice(dataBase, maxit=0)
 meth = init$method
 predM = init$predictorMatrix

 meth[c(nameVar)]="polyreg"

 set.seed(103)
 imputed = mice(dataBase, method=meth, predictorMatrix=predM, m=5)
 imputed <- complete(imputed)

 write.csv(imputed[nameVar], file = paste("dataImp/test/", nameVar, ".csv", sep = ""), row.names =
FALSE)
 return(imputed[nameVar])
}

imputedAll = function(dataBase){
 notNA_NA = detectNA(dataBase)
 notNAvar = notNA_NA[[1]]
 NAvar = notNA_NA[[3]]
 imputedDataBase = dataBase

 for(i in 1:(length(NAvar))){
   cut = sort(c(notNAvar, NAvar[i]))
   dataBaseCut = dataBase[, cut]
   cat(names(NAvar)[i], NAvar[i], "\n")
   imp = imputationMice(dataBaseCut, names(NAvar)[i])
   cat("End imputation")
   imputedDataBase[NAvar[i]] = imp
 }
 return(imputedDataBase)
```

```
}
```

Una vez tratado el problema de los valores perdidos en las variables de nuestra base de datos, se procedió a visualizar la distribución de la variable clase. Como se puede ver en el siguiente histograma la base de datos está bastante desbalanceada. La clase *Colision_Vehiculos* conforma el 55 % de las instancias de la base de datos, hecho que posiblemente empeore la calidad de las predicciones en el futuro.



Una vez visualizado y organizado los datos se procedió a realizar una selección de características de las variables de nuestra base de datos. Esto se hizo en primer lugar para evitar tener que estar tratando con variables que no aportan información de cara al aprendizaje, ya que en teoría algunos algoritmos de aprendizaje funcionarán mejor (más rápido, con menos memoria), y en segundo lugar porque los modelos serán más fácilmente interpretables al ser más simples. Para realizar esta tarea nos apoyamos en el paquete llamado FSelector el cual dispone de una serie de algoritmos capaces de realizar una primera extracción de características e identificar las variables, en un principio, más relevantes de cara a la clasificación en algunos modelos:

```
weights <- FSelector::chi.squared(TIPO_ACCIDENTE~.,dataTrainImp)
sub = FSelector::cutoff.k(weights, 12)
f = as.simple.formula(sub, "TIPO_ACCIDENTE")
print(f)
```

3. Construcción de modelos

Una vez realizado el preprocesamiento necesario entramos de lleno en las tareas de construcción y experimentación de los distintos modelos encargados de realizar la clasificación.

Se trabajo con distintos modelos(Arboles de decision, Técnicas ensambled, svm .. etc) que detallaremos mas adelante. Pero para empezar es necesario explicar cómo se realizan las particiones que se utilizarán luego en el entrenamiento de los modelos. Dichos modelos deberían ser evaluados sobre un conjunto de muestras no usadas ni para aprender el modelo ni para ajustar sus parámetros. Esta sería la única forma de obtener una medida de eficiencia sin sesgo. En esta práctica se utilizó la técnica conocida como *k-validación cruzada*. Esta forma de validación implica el particionado del conjunto de datos en k subconjuntos de igual tamaño (aproximadamente) y obtener k modelos diferentes dejando una partición, de forma sucesiva, como conjunto de test. Es decir, en todos los modelos se usan k − 1 particiones para entrenamiento y 1 para test. La estimación final ofrecida sería la media de las estimaciones para cada uno de los k modelos.

```
k.cv = 5
dataBase = dataTrainImp # Base da datos imputada
classPosition = length(names(dataBase))
classFormula = formula(TIPO_ACCIDENTE ~ .)
#classFormula = f # Entrenamiento con selección de características

# Se crean las particiones del conjunto de datos
ind = seq(1,nrow(dataBase),by=1)
trainPartitions = createFolds(ind, k = k.cv, returnTrain = TRUE)

# Generamos de la misma forma las particiones de test
testPartitions = list()
for(i in 1:k.cv){
 testPartitions[[i]] = ind[-trainPartitions[[i]]]
}
```

Una vez establecido como se realizan las particiones de la base de datos, el siguiente paso fue entrenar los modelos. En esta práctica se han probado el rendimiento de 4 modelos distintos: el árbol de decisión, las Máquinas de Soporte Vectorial (SVM) y los métodos ensamble Random Forest y Bagging.

```
acc = 0
# Bucle de generación de modelos
for(i in 1:k.cv){
 cat("k =", i, "\n")
 # Modelos
 # model = ctree(classFormula, dataBase[trainPartitions[[i]], ])
 model = svm(classFormula, data = dataBase[trainPartitions[[i]], ], cost = 0.5, gamma = 0.001)
 # model = randomForest(classFormula, dataBase[trainPartitions[[i]], ], ntree = 1000)
 # model = adabag::boosting(classFormula, data = dataBase[trainPartitions[[i]], ],
 #                          mfinal = 20,
 #                          control = rpart::rpart.control(maxdepth = 5, minsplit = 5))

 # Test
 testPred = predict(model, newdata = dataBase[testPartitions[[i]], ])

 # Cálculo de las medidas de precisión
```

```
 results = confusionMatrix(table(testPred, dataBase[testPartitions[[i]], classPosition]))
 print(results$overall[[1]])
 acc = acc + results$overall[[1]]

}
print(acc/k.cv)
```

Los resultados y las pruebas más representativas se reflejan en la siguiente tabla. Para evaluar los modelos nos hemos apoyado en la medida de Accuracy, y si nos fijamos en los resultados parece ser que los modelos ofrecen rendimientos similares. El objetivo del entrenamiento de los distintos modelos fue evitar el sobreajuste de los datos y en ello consistió la experimentación con sus hiperparametros.

Modelo	Parámetros	Acc. Local	Acc. Kaggle
Arb. Decisión	Todas la variables	0.82631	0.82217
Abr. Decisión	Selección de características	0.82616	0.82207
svm	Todas la variables, cost = 1, gamma = auto(0.008928571)	0.82147	0.82138
svm	Seleccion de caracteristicas, cost = 1, gamma = auto(0.02222222)	0.82097	0.82089
svm	Seleccion de caracteristicas, cost = 1 gamma = 0.1	0.81957	0.81891
randomForest	Todas las variables, ntree = 500	0.82894	0.82395
randomForest	Todas las variables, ntree = 1000	0.83044	0.82889
randomForest	Seleccion de caracteristicas, ntree = 1000	0.82954	0.82474
bagging	Todas las variables	0.81914	0.81891
bagging	Seleccion de caracteristicas	0.81669	0.81437

4. Conclusiones

Se han aplicado distintas técnicas de preprocesamiento de los datos donde se hizo más hincapié en la imputación de los valores perdidos y en la extracción de características. En cuanto a la construcción de los modelos, se podría considerar que el modelo que mejor resultados ofrece es el Random Forest, aunque también ha sido el modelo que más tiende a sobreajustarse a los datos.

Probablemente el hecho de no haber conseguido mejores tasas de acierto puede ser debido al desbalance de los datos mostrado en los apartados anteriores que provocaba que los modelos cometiecen más fallos en las clases minoritarias y por tanto una disminución en el porcentaje de acierto.