

# Implementación y Análisis de Técnicas Híbridas de Aprendizaje Automático en la Detección Intrusos en Redes de Computadoras

*Deyban Andrés Pérez Abreu*

## Introducción

El presente documento recopila las actividades realizadas en la elaboración del **Trabajo Especial de Grado** de mi persona (autor del documento). Este tiene como tema el **Análisis e Implementación de Técnicas Híbridas de Aprendizaje Automático en la Detección de Intrusos en Redes de Computadoras** haciendo uso del conjunto de datos **NSL-KDD**.

Los objetivos que se buscan lograr con este trabajo es la implementación y análisis de modelos **basados en la firma del ataque**, cómo lo son las **redes neuronales y máquinas de soporte vectorial** en conjunto con técnicas **basadas en anomalías** cómo lo es **K-Medias**. La idea de esta mezcla de paradigmas es la complementación de estos con la esperanza de mejorar el rendimiento desde un punto de vista de **eficacia** a la hora de detectar anomalías en una red de computadoras, específicamente, con la utilización de técnicas basadas en anomalías se busca detectar aquellos ataques conocidos que fueron provistos en el conjunto de entrenamiento, y con las técnicas basadas en anomalías se busca capturar aquellas nuevas anomalías que no fueron provistas en la fase de entrenamiento a los modelos.

El conjunto de datos **NSL-KDD** consta de un conjunto de entrenamiento y un conjunto de prueba excluyentes, es decir, que ningún registro está duplicado entre conjuntos. Adicionalmente, el conjunto de datos de prueba posee ataques que no son proporcionados en el conjunto de entrenamiento, así que la idea es evaluar la capacidad de generalización de los modelos creados simulando un ambiente real de prueba, donde nuevos ataques surgen constantemente.

La tareas a realizar se pueden dividir en tres grandes grupos que se mencionarán a continuación:

1. La primera fase corresponde al pre-procesamiento de los datos, esto aplica tanto al conjunto de entrenamiento como al conjunto de prueba. En este paso se busca crear una vista minable que facilite la manipulación de la información y estandarice los tipos de datos a ser utilizados a lo largo de la investigación.
2. La segunda fase corresponde a la demostración de la eficacia de la propuesta planteada con anterioridad, es decir, la prueba de los modelos híbridos a la hora de realizar las tareas de detección. Esta fase se dividirá en dos conjuntos.
  - **Análisis sobre el conjunto de entrenamiento:** acá se realizarán las pruebas extrayendo un subconjunto de los datos para la prueba y el restante para el entrenamiento y se evaluará el rendimiento de cada uno de los modelos.
  - **Análisis sobre el conjunto de prueba:** acá se tomará el conjunto de entrenamiento en su totalidad para realizar las tareas de entrenamiento y se hará la prueba sobre el conjunto total de prueba provisto por el conjunto de datos NSL-KDD.

**NOTA:** En esta fase los modelos serán entrenados haciendo uso de parámetros por defecto.

3. La tercera fase corresponde al proceso de selección de características y selección de parámetros, en esta fase se analizan los resultados obtenidos del proceso de reducción de características y ajuste de los parámetros para los modelos.

## Pre-Procesamiento de los datos

En esta sección se listarán las actividades realizadas concernientes al proceso de pre-procesamiento de los datos. Esta tarea aplica para los conjuntos de datos de entrenamiento y de prueba, debido a que ambos conjuntos de datos deben poseer el mismo formato para poder realizar el proceso de aprendizaje automático.

Comenzaremos con la configuración del ambiente de trabajo, donde se eliminarán las variables del ambiente de trabajo. Y se cargará un archivo con funciones llamado **functions.R**, este archivo posee una leyenda donde se explica a cabalidad el funcionamiento de cada una de las funciones ilustradas en dicho documento.

```
rm(list = ls())
source("../source/functions/functions.R")
```

A continuación se cargarán los conjuntos de prueba y de entrenamiento a ser utilizados.

```
dataset.training = read.csv(file = "../dataset/KDDTrain+.txt", sep = ",",
header = FALSE)
dataset.testing = read.csv(file = "../dataset/KDDTest+.txt", sep = ",",
header = FALSE)
```

En la variable **dataset.training** se encuentra cargado el conjunto de entrenamiento y en la variable **dataset.testing** se tiene cargado el conjunto de prueba. Veamos las dimensiones de los conjuntos de datos.

```
dim(dataset.training)
```

```
## [1] 125973      43
```

```
dim(dataset.testing)
```

```
## [1] 22544      43
```

El **conjunto de entrenamiento** tiene 125973 filas y 43 columnas. Por otra parte, el **conjunto de prueba** tiene 22544 filas y 43 columnas. Es importante mencionar que de las 43 columnas, la **columna 42** corresponde a la etiqueta del ataque y la **columna 43** corresponde a la cantidad de clasificadores que acertaron a la hora de clasificar dicho registro en el proceso de creación del conjunto de datos NSL-KDD. En el proceso previamente mencionado se utilizaron 21 clasificadores, por dicho motivo, el rango de valores en esta columna está comprendido entre [0,21]. A continuación veamos si los conjuntos de datos poseen valores faltantes, para ello haremos uso de la función **complete.cases**.

```
sum(complete.cases(dataset.training)) == nrow(dataset.training)
```

```
## [1] TRUE
```

```
sum(complete.cases(dataset.testing)) == nrow(dataset.testing)
```

```
## [1] TRUE
```

Se observa que la cantidad de casos completos es igual a la cantidad de filas de ambos conjuntos de datos, por tal motivo no existen valores faltantes. Ahora veamos los tipos de ataques por conjuntos de datos. Empezaremos por con el conjunto de entrenamiento.

```
attacks.training = unique(dataset.training$V42)
attacks.training = sort(as.character(attacks.training))
length(attacks.training)
```

```
## [1] 23
```

Se observa que el conjunto de entrenamiento consta de 23 etiquetas, donde 1 corresponde a la etiqueta de **tráfico normal**, y las otras 22 corresponden a **ataques**. Ahora veamos el conjunto de prueba.

```
attacks.testing = unique(dataset.testing$V42)
attacks.testing = sort(as.character(attacks.testing))
length(attacks.testing)
```

```
## [1] 38
```

Se observan 38 etiquetas en el conjunto de prueba, donde 1 corresponde a la etiqueta de **tráfico normal** y las otras 37 corresponden a **ataques**. En este punto se puede observar cómo hay mayor cantidad de ataques en el conjunto de prueba que en el conjunto de entrenamiento, esto es debido a que el conjunto de prueba busca medir la habilidad del modelo de **aprendizaje automático** para generalizar ante ataques no vistos en el conjunto de entrenamiento con anterioridad. A continuación se observan cuales son los ataques presentes en el conjunto de prueba que no están presentes en el conjunto de prueba y viceversa. Se empezará con examinar la cantidad total de ataques presentes entre ambos conjuntos.

```
total.attacks = sort(unique(c(attacks.training, attacks.testing)))
length(total.attacks)
```

```
## [1] 40
```

Entre ambos conjuntos se observan 40 etiquetas, donde 1 corresponde al **tráfico normal** y las otras 39 corresponden a etiquetas de **ataques**. De lo anterior se puede concluir que hay 17 tipos de ataques presentes en el conjunto de prueba que no están presentes en el conjunto de entrenamiento, y que hay 2 tipos de ataques en el conjunto de entrenamiento que no están presentes en el conjunto de prueba. A continuación se listarán aquellas etiquetas comunes entre ambos conjuntos de datos.

```
total.attacks = sort(unique(c(attacks.training, attacks.testing)))
length(total.attacks)
```

```
## [1] 40
```

```
total.attacks
```

```
## [1] "apache2"          "back"           "buffer_overflow"
## [4] "ftp_write"         "guess_passwd"    "httptunnel"
## [7] "imap"              "ipsweep"        "land"
## [10] "loadmodule"        "mailbomb"       "mscan"
## [13] "multihop"          "named"          "neptune"
## [16] "nmap"              "normal"         "perl"
## [19] "phf"               "pod"            "portsweep"
## [22] "processtable"     "ps"             "rootkit"
## [25] "saint"             "satan"          "sendmail"
```

```

## [28] "smurf"           "snmpgetattack"   "snmpguess"
## [31] "spy"              "sqlattack"       "teardrop"
## [34] "udpstorm"        "warezclient"     "warezmaster"
## [37] "worm"             "xlock"          "xsnoop"
## [40] "xterm"

```

Se observa que existen 21 etiquetas comunes entre ambos conjuntos de datos, donde 1 corresponde a la etiqueta de **tráfico normal** y las otras 20 corresponden a **ataques**. Todas las etiquetas fueron listadas. A continuación se listarán aquellos **ataques** que están presentes en el conjunto de prueba y no en el conjunto de entrenamiento.

```

index.attacks = which(attacks.testing %in% attacks.training)
length(attacks.testing[-index.attacks])

```

```
## [1] 17
```

```
attacks.testing[-index.attacks]
```

```

## [1] "apache2"         "httptunnel"      "mailbomb"       "mscan"
## [5] "named"           "processtable"    "ps"            "saint"
## [9] "sendmail"        "snmpgetattack"  "snmpguess"     "sqlattack"
## [13] "udpstorm"       "worm"          "xlock"         "xsnoop"
## [17] "xterm"

```

Son 17 los ataques presentes en el conjunto de prueba que no están presentes en el conjunto de entrenamiento, los mismos fueron listados. A continuación se listarán aquellos ataques presentes en el conjunto de entrenamiento que no lo están en el conjunto de prueba.

```

index.attacks.training = which(attacks.training %in% attacks.testing)
length(attacks.training[-index.attacks.training])

```

```
## [1] 2
```

```
attacks.training[-index.attacks.training]
```

```
## [1] "spy"              "warezclient"
```

Son sólo 2 los ataques en el conjunto de entrenamiento que no están presentes en el conjunto de prueba. Estos corresponden a **spy** y **warezclient**.

## Extracción de características

En este documento se clasifican las anomalías en 4 grupos **DoS**, **Probing**, **R2L** y **U2R**, es decir, habrán 5 etiquetas, donde 4 corresponden a los tipos de ataques mencionados previamente y la 5ta etiqueta corresponde a la etiqueta normal.

Para facilitar el trabajo se debe asociar cada uno de los ataques a cada una de las clases mencionadas con anterioridad. Para esto se hará uso de la función **ClassLabelAttack** que recibe como parámetro un **dataframe** y retorna una columna con la clase de cada tipo de ataque para cada registro. Estos nombres colocados acordes a la investigación hecha por **Bhavsar**.

```
dataset.training$V44 = ClassLabelAttack(dataset.training)
dataset.testing$V44 = ClassLabelAttack(dataset.testing)
```

De esta manera, tanto el **conjunto de entrenamiento** como el **conjunto de prueba** tienen una nueva columna en la que cada registro tiene asociada la respectiva clase a la que pertenece. Adicionalmente se agregó una nueva columna que corresponde a una nueva etiqueta que identifica a cada registro como **ataque** o **normal**. De esta manera se tiene una clase general para la asociación de los registros.

```
dataset.training$V45 = NormalAttackLabel(dataset.training)
dataset.testing$V45 = NormalAttackLabel(dataset.testing)
```

Ahora se dividirá el conjunto de datos en **dataframes** individuales para cada clase: **DoS**, **normal**, **R2L**, **U2R**.

```
training.split = split(dataset.training, dataset.training$V44)
testing.split = split(dataset.testing, dataset.testing$V44)
summary(training.split)
```

```
##          Length Class      Mode
## DoS        45   data.frame  list
## normal     45   data.frame  list
## Probing    45   data.frame  list
## R2L        45   data.frame  list
## U2R        45   data.frame  list

summary(testing.split)
```

```
##          Length Class      Mode
## DoS        45   data.frame  list
## normal     45   data.frame  list
## Probing    45   data.frame  list
## R2L        45   data.frame  list
## U2R        45   data.frame  list
```

Las variables **training.split** y **testing.split** contienen una lista de sub-conjuntos por etiquetas de las clases de los ataques en ambos conjuntos de datos. A continuación se listará el número de cada clase en el conjunto de entrenamiento.

```
nrow(training.split$DoS)
```

```
## [1] 45927
```

```
nrow(training.split$normal)
```

```
## [1] 67343
```

```
nrow(training.split$Probing)
```

```
## [1] 11656
```

```
nrow(training.split$R2L)
```

```
## [1] 995
```

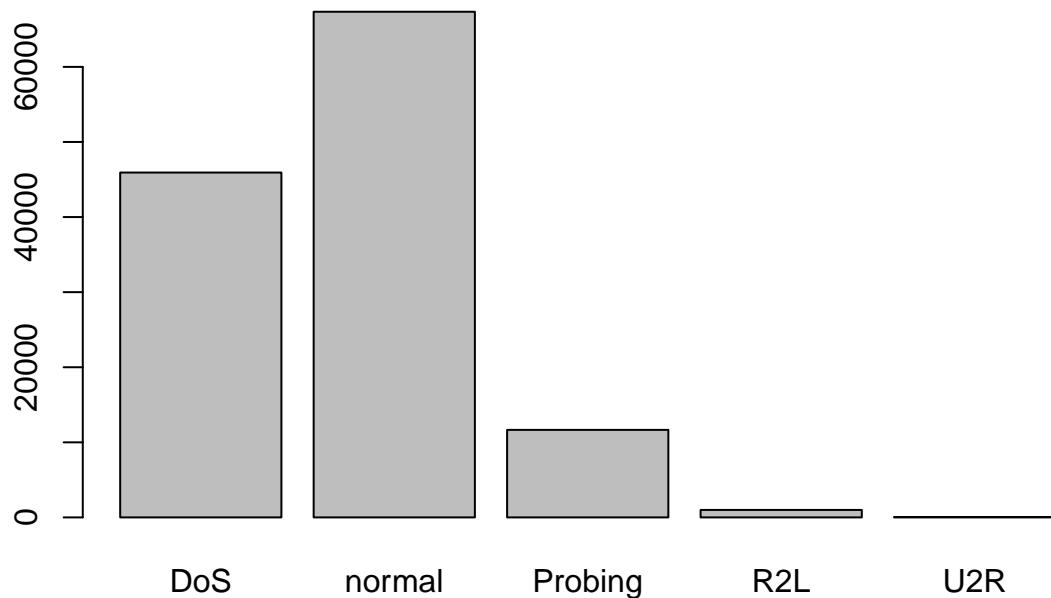
```
nrow(training.split$U2R)
```

```
## [1] 52
```

Se observa que la clase **normal** es la que más registros posee en el conjunto de datos de entrenamiento, seguido por la clase **DoS**. Lo anterior nos da una idea de cuáles son las clases de ataques más comunes y menos comunes. A continuación se presenta un gráfico que ilustra lo anterior y permite visualizar mejor la distribución de las clases.

```
barplot(table(dataset.training$V44), main = "Frecuencia de las Clases en el Conjunto  
de Entrenamiento")
```

**Frecuencia de las Clases en el Conjunto  
de Entrenamiento**



A continuación se repiten los pasos anteriores para el conjunto de prueba.

```
nrow(testing.split$DoS)
```

```
## [1] 7458
```

```
nrow(testing.split$normal)
```

```
## [1] 9711
```

```
nrow(testing.split$Probing)
```

```
## [1] 2421
```

```
nrow(testing.split$R2L)
```

```
## [1] 2754
```

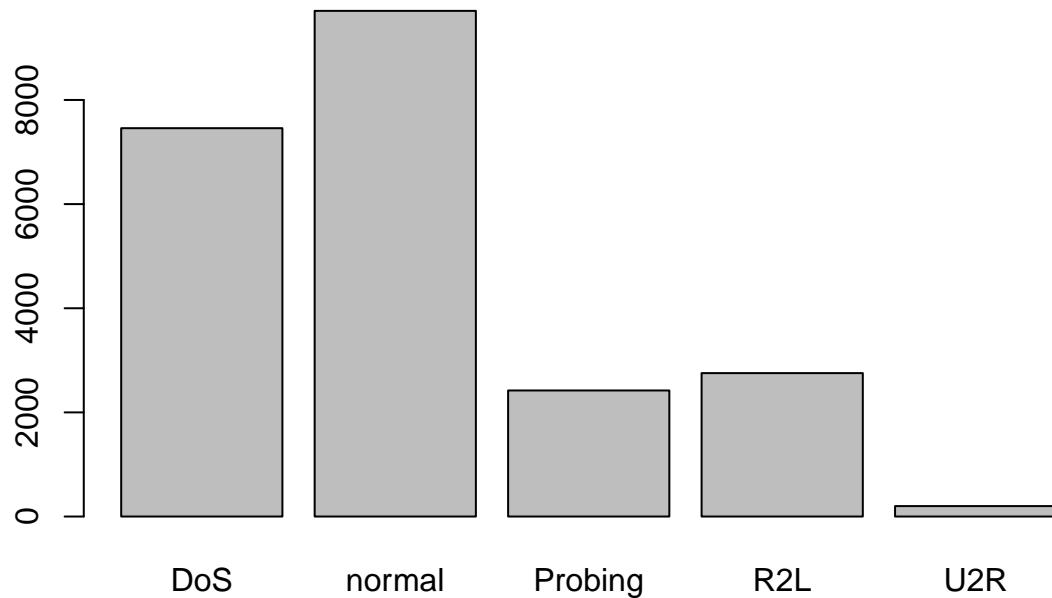
```
nrow(testing.split$U2R)
```

```
## [1] 200
```

En esta oportunidad la clase **normal** sigue siendo la clase con mayor cantidad de registros. En contraste con el conjunto de entrenamiento, se observa que en esta ocasión las clases **Probing** y **R2L** están más equilibradas, adicionalmente, la clase **U2R** posee una cantidad mucho mayor de registros que en el conjunto de entrenamiento. A continuación se presenta un gráfico con las distribuciones de las clases en el conjunto de prueba.

```
barplot(table(dataset.testing$V44), main = "Frecuencia de las Clases en el Conjunto  
de Prueba")
```

**Frecuencia de las Clases en el Conjunto  
de Prueba**



### Renombramiento de las columnas

Se hará uso de la función **ColumnNames** que asigna a los conjuntos de datos los nombres respectivos, estos nombres colocados acordes a la investigación hecha por **Bhavsar**.

```
dataset.training = ColumnNames(dataset.training)
dataset.testing = ColumnNames(dataset.testing)
```

## Eliminación de características no importantes

En esta sección se examinarán posibles características inútiles, esto es, aquellas características que sólo tienen un nivel de valores, por ejemplo, una característica de tipo **numérico** donde en todos los registros el valor es cero (0), es decir, el rango viene dado por [0]. Para dicho propósito se utilizará la función **CheckFeaturesLevels** que toma como entrada un dataframe y retorna la posición (si existe) de la característica que no aporta información.

```
index.dummy.variables.training = CheckFeaturesLevels(dataset.training)
index.dummy.variables.testing = CheckFeaturesLevels(dataset.testing)
names(dataset.training)[index.dummy.variables.training]
```

```
## [1] "Num_outbound_cmds"
names(dataset.testing)[index.dummy.variables.testing]
```

```
## [1] "Num_outbound_cmds"
```

Se observa que en ambos conjuntos de datos la columna **Num\_outbound\_cmds** es inútil, en consecuencia, la misma será eliminada del conjunto de datos.

```
dataset.training[,index.dummy.variables.training] = NULL
dataset.testing[, index.dummy.variables.testing] = NULL
```

## Transformación de los datos

Las columnas **Protocol\_type**, **Service** y **Flag** tienen tipos de datos categóricos, los mismos serán transformados a numéricos. La transformación tiene su justificación en el hecho de que los algoritmos a utilizar que son **Redes Neuronales**, **Máquinas de Soporte Vectorial** y **K-Medias** funcionan con variables predictoras (características) numéricas. Dicho esto es obligatorio transformar las columnas de tipo categórico a tipo numérico.

1. **Protocol\_type**: esta característica posee 3 niveles, que serán listados alfabéticamente a continuación.

```
sort(unique(dataset.training$Protocol_type))
```

```
## [1] icmp tcp udp
## Levels: icmp tcp udp
```

```
sort(unique(dataset.testing$Protocol_type))
```

```
## [1] icmp tcp udp
## Levels: icmp tcp udp
```

Los mismos se transformarán en los valores 1,2,3 respectivamente. La función **ProtocolTransformation** es la encargada de realizar dicho trabajo.

```
dataset.training = ProtocolTransformation(dataset.training)
dataset.testing = ProtocolTransformation(dataset.testing)
```

2. **Service:** esta característica posee una mayor cantidad de niveles con respecto a **Protocol\_type**, los mismo serán listados a continuación.

```
sort(unique(dataset.training$Service))
```

```
## [1] aol      auth     bgp      courier  csnet_ns
## [6] ctf      daytime  discard   domain   domain_u
## [11] echo    eco_i    ecr_i    efs      exec
## [16] finger   ftp      ftp_data  gopher   harvest
## [21] hostnames http    http_2784 http_443 http_8001
## [26] imap4    IRC      iso_tsap klogin   kshell
## [31] ldap     link    login    mtp      name
## [36] netbios_dgm netbios_ns netbios_ssn netstat  nnsp
## [41] nntp     ntp_u   other    pm_dump pop_2
## [46] pop_3    printer  private  red_i   remote_job
## [51] rje      shell   smtp    sql_net ssh
## [56] sunrpc   supdup  systat  telnet  tftp_u
## [61] time     tim_i   urh_i   urp_i   uucp
## [66] uucp_path vmnet   whois   X11     Z39_50
## 70 Levels: aol auth bgp courier csnet_ns ctf daytime discard ... Z39_50
```

```
sort(unique(dataset.testing$Service))
```

```
## [1] auth     bgp      courier  csnet_ns  ctf
## [6] daytime  discard   domain   domain_u echo
## [11] eco_i    ecr_i    efs      exec     finger
## [16] ftp      ftp_data  gopher   hostnames http
## [21] http_443 imap4    IRC      iso_tsap klogin
## [26] kshell   ldap     link    login    mtp
## [31] name    netbios_dgm netbios_ns netbios_ssn netstat
## [36] nnsp     nntp     ntp_u   other    pm_dump
## [41] pop_2    pop_3    printer  private  remote_job
## [46] rje      shell   smtp    sql_net ssh
## [51] sunrpc   supdup  systat  telnet  tftp_u
## [56] time     tim_i   urp_i   uucp    uucp_path
## [61] vmnet   whois   X11     Z39_50
## 64 Levels: auth bgp courier csnet_ns ctf daytime discard ... Z39_50
```

Se observa que en el **conjunto de entrenamiento** hay un total de 70 niveles, contra 64 niveles presentes en el **conjunto de prueba**. Observemos la cantidad total de servicios uniendo ambos conjuntos.

```
sort(unique(dataset.training$Service))
```

```
## [1] aol      auth     bgp      courier  csnet_ns
## [6] ctf      daytime  discard   domain   domain_u
## [11] echo    eco_i    ecr_i    efs      exec
## [16] finger   ftp      ftp_data  gopher   harvest
```

```

## [21] hostnames http http_2784 http_443 http_8001
## [26] imap4 IRC iso_tsap klogin kshell
## [31] ldap link login mtp name
## [36] netbios_dgm netbios_ns netbios_ssn netstat nnsp
## [41] nntp ntp_u other pm_dump pop_2
## [46] pop_3 printer private red_i remote_job
## [51] rje shell smtp sql_net ssh
## [56] sunrpc supdup systat telnet tftp_u
## [61] time tim_i urh_i urp_i uucp
## [66] uucp_path vmnet whois X11 Z39_50
## 70 Levels: aol auth bgp courier csnet_ns ctf daytime discard ... Z39_50

```

Se observa que el total de servicios es de 70, es decir, el conjunto de servicios en el conjunto de entrenamiento corresponde al universo de todos los servicios en los conjuntos de datos.

Los niveles serán enumerados en un rango [1,70] en orden alfabético, tal como se muestra a continuación.

```
sort(unique(c(as.character(unique(dataset.testing$Service)),
           as.character(unique(dataset.training$Service)))))
```

```

## [1] "aol"      "auth"     "bgp"      "courier"   "csnet_ns"
## [6] "ctf"       "daytime"   "discard"   "domain"    "domain_u"
## [11] "echo"      "eco_i"     "ecr_i"     "efs"       "exec"
## [16] "finger"    "ftp"       "ftp_data"   "gopher"    "harvest"
## [21] "hostnames" "http"      "http_2784"  "http_443"   "http_8001"
## [26] "imap4"     "IRC"       "iso_tsap"   "klogin"    "kshell"
## [31] "ldap"      "link"      "login"     "mtp"       "name"
## [36] "netbios_dgm" "netbios_ns" "netbios_ssn" "netstat"   "nnsp"
## [41] "nntp"      "ntp_u"     "other"     "pm_dump"   "pop_2"
## [46] "pop_3"      "printer"   "private"   "red_i"     "remote_job"
## [51] "rje"        "shell"     "smtp"      "sql_net"   "ssh"
## [56] "sunrpc"    "supdup"    "systat"   "telnet"    "tftp_u"
## [61] "time"       "tim_i"     "urh_i"     "urp_i"     "uucp"
## [66] "uucp_path" "vmnet"    "whois"    "X11"      "Z39_50"

```

Se utilizará la función **ServiceTransformation** para enumerar cada uno de los servicios listados previamente.

```
dataset.training = ServiceTransformation(dataset.training)
dataset.testing = ServiceTransformation(dataset.testing)
```

3. **Flag:** es la característica categórica restante. Observemos los niveles de esta característica.

```
sort(unique(dataset.training$Flag))
```

```

## [1] OTH   REJ   RST0   RSTOSO RSTR   S0    S1    S2    S3    SF
## [11] SH
## Levels: OTH REJ RST0 RSTOSO RSTR S0 S1 S2 S3 SF SH

```

```
sort(unique(dataset.testing$Flag))
```

```

## [1] OTH   REJ   RST0   RSTOSO RSTR   S0    S1    S2    S3    SF
## [11] SH
## Levels: OTH REJ RST0 RSTOSO RSTR S0 S1 S2 S3 SF SH

```

```
length(sort(unique(c(as.character(unique(dataset.testing$Flag)),  
as.character(unique(dataset.training$Flag))))))
```

```
## [1] 11
```

Se observa que hay 11 niveles en ambos conjuntos y que la unión de los niveles de ambos conjuntos de datos arroja el mismo resultado. Dicho esto, las etiquetas serán enumeradas por orden alfabético, tal como se muestra a continuación.

```
sort(unique(c(as.character(unique(dataset.testing$Flag)),  
as.character(unique(dataset.training$Flag)))))
```

```
## [1] "OTH"      "REJ"      "RSTO"     "RSTOSO"   "RSTR"     "S0"       "S1"  
## [8] "S2"       "S3"       "SF"        "SH"
```

Se utilizará la función **FlagTransformation** para dicho propósito.

```
dataset.training = FlagTransformation(dataset.training)  
dataset.testing = FlagTransformation(dataset.testing)
```

## Guardando la vista minable

En este punto la vista minable ya fue creada, las columnas poseen un formato adecuado para los algoritmos que serán utilizados y se agregaron nuevas columnas que facilitarán tareas futuras en la investigación. Debido a que no hay más tareas por hacer, se procede a guardar los conjuntos de datos para cargar los datos preprocesados y no tener que repetir dicho procedimiento luego.

```
write.csv(dataset.training,  
         file = "../dataset/NSLKDD_Training_New.csv", row.names = FALSE)  
write.csv(dataset.testing,  
         file = "../dataset/NSLKDD_Testing_New.csv", row.names = FALSE)
```

## Implementación de modelos híbridos

La propuesta del **trabajo especial de grado** consta del entrenamiento de dos modelos híbridos de aprendizaje automático. El primer modelo (i) consta de una **red neuronal** en el primer nivel y **K-Medias** en el segundo, por otra parte, el segundo modelo (ii) consta de una **máquina de soporte vectorial** en el primer nivel y **K-Medias** en el segundo nivel.

En esta sección los modelos serán entrenados con los parámetros por defecto. Posteriormente se hará selección de características y parámetros, y se analizará el impacto con respecto a los modelos creados en esta sección.

Esta sección será dividida en dos grandes secciones, una concerniente al entrenamiento y evaluación de los modelos utilizando el **conjunto de entrenamiento** exclusivamente. En esta fase se hará uso de la **técnica de validación cruzada de 10 conjuntos** para evaluar los modelos. Posteriormente, se hará el entrenamiento haciendo uso total del conjunto de entrenamiento y se hará evaluación de los modelos haciendo uso del conjunto de prueba.

Con la estrategia descrita en el párrafo anterior se podrán observar tres aspectos relevantes:

1. La eficacia de los modelos contra ataques conocidos.
2. La eficacia de los modelos contra ataques no conocidos.
3. Diferencias de rendimiento entre ambos modelos.

## Análisis sobre el conjunto de entrenamiento

En esta sección se listarán las actividades concernientes al entrenamiento y evaluación de los modelos híbridos haciendo uso exclusivo del conjunto de entrenamiento y de la técnica de **validación cruzada de 10 conjuntos** para la evaluación de los modelos.

Inicialmente iniciaremos con una evaluación del rendimiento de **K-Medias**, se elegirán los centroides y se evaluará su desempeño en la tarea de detección de intrusos.

### K-Medias

Se empezará por establecer el ambiente de trabajo eliminando las variables parciales, cargando el archivo de funciones y la vista minable del conjunto de entrenamiento pre-procesado previamente.

```
rm(list = ls())
source("../source/functions/functions.R")
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv",
                             sep = ",", header = TRUE)
```

Para esta sección sólo se necesitaran las etiquetas **Label\_Normal\_ClassAttack** y **Label\_Normal\_or\_Attack**, por lo tanto las otras etiquetas serán eliminadas.

```
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
```

A continuación se le asignará el tipo numérico a todas las **columnas predictoras**, y el tipo factor a las **columnas de etiquetas**.

```
for (i in 1 : (ncol(dataset) -2) )
  dataset[,i] = as.numeric(dataset[,i])

for (i in (ncol(dataset) -1):ncol(dataset) )
  dataset[,i] = as.factor(dataset[,i])
```

En este punto se crearán dos nuevos conjuntos de datos, **dataset.two** que tendrá como etiqueta la columna **Label\_Normal\_or\_Attack**, columna que sólo tiene dos niveles categóricos **Attack** o **normal**. Por otra parte, se creará un segundo conjunto de datos llamado **dataset.five** el cuál contendrá como etiqueta la columna **Label\_Normal\_ClassAttack**, columna que tiene cinco niveles categóricos **DoS**, **normal**, **Probing**, **R2L** y **U2R**.

```
dataset.two = dataset[ -(ncol(dataset)-1)]
dataset.two[, ncol(dataset.two)] = as.character(dataset.two[, ncol(dataset.two)])
dataset.five = dataset[-ncol(dataset)]
```

Hasta este punto se tienen tres conjuntos de datos **dataset**, **dataset.two** y **dataset-five**. El algoritmo **K-Medias** funciona utilizando medidas de distancia, motivo por el cual es necesario el escalamiento de los valores de las columnas predictoras dentro de un mismo rango de valores, de lo contrario el rendimiento del algoritmo se verá deteriorado por la diferencia entre las escalas. La función **ScaleSet** lleva todas las columnas a un rango de valores con *media* cero (0), y *desviación estándar* uno (1). Adicionalmente, del conjunto de datos **dataset** se eliminará la columna **Label\_Normal\_or\_Attack** debido a que ya no es necesaria.

```

dataset$Label_Normal_or_Attack = NULL
dataset = ScaleSet(dataset)
dataset.two = ScaleSet(dataset.two)
dataset.five = ScaleSet(dataset.five)

```

## Codo de Jambu

Hasta este punto ya se tienen los conjuntos de datos listos para ser utilizados. El algoritmo **K-Medias** amerita que se le pasen como argumentos el número de centroides o la posición inicial de los centroides. Estos corresponden al número de conjuntos que se esperan identificar en el conjunto de datos. En el escenario que tenemos actualmente esta tarea es sencilla debido a que el conjunto de datos tiene cada uno de los registros etiquetados, sin embargo, este paso no siempre es sencillo. Adicionalmente es importante recordar que el algoritmo de **K-Medias** es un algoritmo de enfoque **no-supervisado** y no hace uso de estas etiquetas para separar los conjuntos.

Si bien es cierto que tenemos etiquetas que nos dicen de antemano cuáles son los niveles de los conjuntos, existe un dilema con respecto al rendimiento del algoritmo con **dos** o **cinco** clases objetivo. El problema de la selección del número de clusters siempre ha existido y existe un método llamado **Codo de Jambu** que es utilizado para capturar la varianza acumulada con respecto al número de clusters usados. Al graficar la cantidad de varianza acumulada por el número de clusters, se verá que llega un punto en el que la gráfica tiene un comportamiento que emula la articulación de un codo, y es en ese punto, donde se empieza a formar la articulación, es donde se indica que el uso de mayor cantidad de clusters no aporta mayor información al algoritmo.

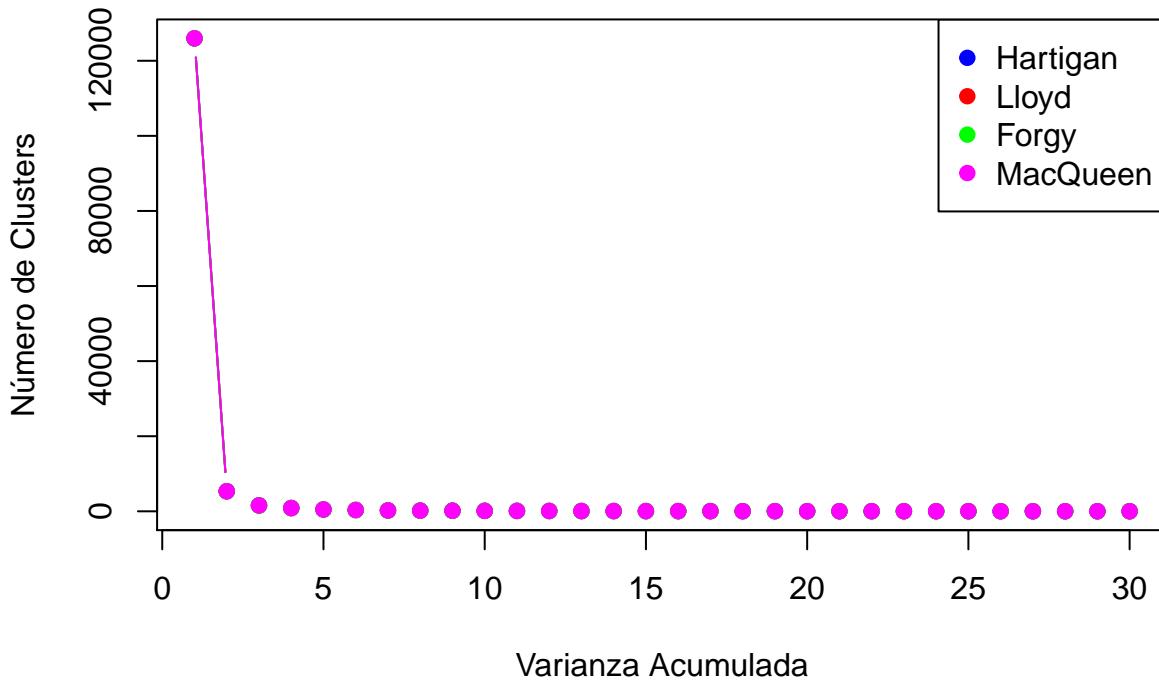
Existen cuatro tipos de algoritmos para calcular las distancias de K-Medias, estas son **Hartigan-Wong**, **Lloyd**, **Forgy** y **Macqueen**. A continuación se aplicará cada una de estas técnicas variando la cantidad de centroides en el rango [1,30], se graficarán los resultados y se analizarán los mismos.

```

IIC.Hartigan = vector(mode = "numeric", length = 30)
IIC.Lloyd = vector(mode = "numeric", length = 30)
IIC.Forgy = vector(mode = "numeric", length = 30)
IIC.MacQueen = vector(mode = "numeric", length = 30)
for (k in 1:30)
{
  groups = kmeans(dataset[,ncol(dataset)-2], k, iter.max = 100,
                  algorithm = "Hartigan-Wong")
  IIC.Hartigan[k] = groups$tot.withinss
  groups = kmeans(dataset[,ncol(dataset)-2], k, iter.max = 100, algorithm = "Lloyd")
  IIC.Lloyd[k] = groups$tot.withinss
  groups = kmeans(dataset[,ncol(dataset)-2], k, iter.max = 100, algorithm = "Forgy")
  IIC.Forgy[k] = groups$tot.withinss
  groups = kmeans(dataset[,ncol(dataset)-2], k, iter.max = 100, algorithm = "MacQueen")
  IIC.MacQueen[k] = groups$tot.withinss
}
plot(IIC.Hartigan, col = "blue", type = "b", pch = 19, main = "Codo de Jambu",
      xlab = "Varianza Acumulada", ylab = "Número de Clusters")
points(IIC.Lloyd, col = "red", type = "b", pch = 19)
points(IIC.Forgy, col = "green", type = "b", pch = 19)
points(IIC.MacQueen, col = "magenta", type = "b", pch = 19)
legend("topright", legend = c("Hartigan", "Lloyd", "Forgy", "MacQueen"),
       col = c("blue", "red", "green", "magenta"), pch = 19)

```

## Codo de Jambu



Se puede observar como con dos clusters se alcanza el mejor resultado, debido a que cómo se observa en el gráfico, la transición entre la varianza acumulada con dos y tres clusters hace la analogía del codo que corresponde a la articulación mencionada con anterioridad. Adicionalmente se puede observar que todos los algoritmos se solapan entre si, este comportamiento es indicativo de que todos se comportan de manera similar y es indiferente su uso en este conjunto de datos. Para mayor información consultar el siguiente enlace: Codo de Jambu.

En las próximas secciones se probará si estos resultados son ciertos evaluando el rendimiento del algoritmo utilizando 5 clusters y 2 clusters respectivamente.

### K-Medias (5 clusters)

Empezaremos con 5 clusters, debido a que aparentemente es el que tiene peor rendimiento. La metodología es la siguiente, se aplicará 10 veces el algoritmo y se promediará la tasa de acierto para evaluar el desempeño.

```
results.five = vector(mode = "numeric", length = 10)
best.accuracy.five = 0
for (i in 1:length(results.five))
{
  set.seed(i)
  model.kmeans.five = kmeans(dataset.five[,1:(ncol(dataset.five)-1)],
                             5, iter.max = 100)

  prediction.five = OrderKmeans(model.kmeans.five)
  accuracy.five = mean(prediction.five == dataset.five$Label)

  results.five[i] = accuracy.five

  if(best.accuracy.five < accuracy.five)
```

```

{
  best.prediction.five = prediction.five
  best.accuracy.five = accuracy.five
}
}

```

La variable `results.five` contiene los resultados de la tasa de aciertos de cada iteración, y las variables `best.prediction.five` y `best.accuracy.five` contienen las mejores predicciones y la mejor tasa de aciertos respectivamente. Veamos los resultados.

```

results.five * 100

## [1] 69.87688 68.85364 69.35454 77.91590 68.58771 72.29724 76.55132
## [8] 59.90252 78.60891 76.77280

mean(results.five) * 100

## [1] 71.87215

```

Se observa que el promedio de acierto fue de 71, 87%, este rendimiento no parece estar mal, sin embargo, es necesario esperar a la comparación con el modelo de dos clusters para poder tener una mejor opinión. Mientras tanto crearemos una matriz de confusión para ilustrar el desempeño del modelo de manera gráfica.

```

confusion.matrix.five = table(Real = dataset.five$Label, Prediction = best.prediction.five)
confusion.matrix.five

##          Prediction
## Real      DoS normal Probing   R2L   U2R
##   DoS     34329    4691    6888     9    10
##   normal    115   63828    119    425  2856
##   Probing   384    5845    869   4217   341
##   R2L        3     940      0      0    52
##   U2R        0     52       0      0      0

```

La matriz de confusión se ve bastante desordenada, y no acertó en la predicción de ningún registro para las clases **R2L** y **U2R**. Veamos la mejor tasa de acierto y la peor tasa de aciertos.

```
best.accuracy.five*100
```

```
## [1] 78.60891
```

```
best.accuracy.five*100
```

```
## [1] 78.60891
```

La mejor tasa de aciertos fue de 78%, no parece ser un mal rendimiento, pero debemos esperar a la comparación con el otro modelo. Como aspecto importante a resaltar, la diferencia entre los resultados se debe a que la inicialización de los centroides en el algoritmo de **K-Medias** se hace de forma aleatoria, y dependiendo de la posición iniciales de los centroides, el algoritmo puede converger a diferentes mínimos locales. Por tal motivo, se colocaron semillas, de tal manera que las pruebas puedan ser recreables. A continuación calcularemos la eficacia por etiqueta, la salida es un vector numérico que representa a las clases ordenadas en orden alfabético de la siguiente forma: **DoS**, **normal**, **Probing**, **R2L** y **U2R**.

```
AccuracyPerLabel(confusion.matrix.five, dataset.five)
```

```
## [1] 74.746881 94.780452 7.455388 0.000000 0.000000
```

Se aprecia el hecho de que el algoritmo sólo clasifica bien las clases **DoS** y **Normal**. Estas dos clases corresponden a la mayoría de registros del conjunto de datos y por eso es que el promedio de aciertos es elevado, sin embargo, el rendimiento con las otras tres etiquetas **Probing**, **R2L** y **U2R** es pobre.

Lo siguiente será transformar la matriz de confusión de **cinco clases a dos clases**. Esto con la finalidad de poder calcular medidas de rendimiento binarias como lo son **sensibilidad**, **especificidad** y **precisión**.

```
attack.normal.confusion.matrix.five = AttackNormalConfusionMatrix(dataset.five,
                                                               best.prediction.five)
```

```
attack.normal.confusion.matrix.five
```

```
##          Prediction
## Real      Attack normal
##   Attack  47102 11528
##   normal   3515 63828
```

Se observa cómo hay una gran cantidad de **falsos negativos** y **falsos positivos**, a pesar de que la mayoría de los registros son clasificados de buena manera. Ahora veamos la eficacia por etiqueta. La salida corresponde a un vector numérico donde la primera posición es **Attack** y la segunda **normal**.

```
AccuracyPerLabel(attack.normal.confusion.matrix.five, dataset.two)
```

```
## [1] 80.33771 94.78045
```

La eficacia a la hora de detectar tráfico normal es bastante elevada, de 94.78%. Para la detección de los ataques es menor, esta corresponde a 80.34%, que es un número elevado, sin embargo, hay que recordar que este número está sesgado desde el punto de vista que sólo se clasificaron ataques de tipo **DoS**. A continuación se calcularán las medidas de rendimiento binarias.

```
Sensitivity(attack.normal.confusion.matrix.five) * 100
```

```
## [1] 80.33771
```

```
Especifity(attack.normal.confusion.matrix.five) * 100
```

```
## [1] 94.78045
```

```
Precision(attack.normal.confusion.matrix.five) * 100
```

```
## [1] 93.05569
```

Se observa que el modelo es bueno detectando tráfico normal, sin embargo, a la hora de detectar ataques el rendimiento se ve mermado.

## K-Medias (2 clusters)

Ahora se implementarán los pasos realizados con cinco clusters pero ahora con dos clusters. Es decir, se correrá el algoritmo de **K-Medias** diez veces con dos clusters.

```

results.two = vector(mode = "numeric", length = 10)
best.accuracy.two = 0
for (i in 1:length(results.two))
{
  set.seed(i)
  model.kmeans.two = kmeans(dataset.two[,1:(ncol(dataset.two)-1)],
                            2, iter.max = 100)

  prediction.two = OrderKmeans(model.kmeans.two)
  accuracy.two = mean(prediction.two == dataset.two$Label)

  results.two[i] = accuracy.two

  if(best.accuracy.two < accuracy.two)
  {
    best.prediction.two = prediction.two
    best.accuracy.two = accuracy.two
  }
}

```

La variable **resultst.two** contiene la tasa de aciertos en cada iteración del algoritmo.

```

results.two

## [1] 0.9050590 0.9050590 0.9050590 0.9050590 0.8103244 0.6087177 0.4935740
## [8] 0.6087177 0.6087177 0.9050590

```

Se observa que la tasa de aciertos es mayor que con cinco clusters, adicionalmente, hubo iteraciones en la que los resultados se repitieron. Esto es debido a que la inicialización de los centroides al inicio del algoritmo hizo que en esas ocasiones se alcanzara el mismo **mínimo local**. Este comportamiento da indicios de que la solución al conjunto de datos se representa con dos clusters y no con cinco. A continuación calculemos el promedio de acierto.

```

mean(results.two) * 100

## [1] 76.55347

```

La media de acierto es de 76.55%, este resultado es mayor al promedio con cinco clusters. Se creará una matriz de confusión para visualizar gráficamente el desempeño del algoritmo en la tarea de clasificación.

```

confusion.matrix.two = table(Real = dataset.two$Label, Prediction = best.prediction.two)
confusion.matrix.two

##          Prediction
## Real      Attack normal
##   Attack  47351   11279
##   normal   681   66662

```

Se aprecia que contiene una alto número de **falsos negativos**, en realidad es una cantidad similar a la matriz de confusión con cinco clusters. Por otra parte, la cantidad de **falsos positivos** se vio reducida notablemente. Ahora imprimiremos la **tasa de aciertos** y la **tasa de error** del mejor modelo obtenido.

```
best.accuracy.two*100  
  
## [1] 90.5059  
  
ErrorRate(best.accuracy.two)*100  
  
## [1] 9.494098
```

Se obtuvo una **tasa de aciertos** de 90.51% una tasa bastante alta, mucho mayor que el modelo con cinco clusters. Por consecuente, la **tasa de errores** será también menor. Ahora veamos la **tasa de aciertos** por etiquetas. Es importante recordar que la salida corresponde a un vector numérico donde la primera posición corresponde a la etiqueta **Attack** y la segunda a la etiqueta **normal**.

```
AccuracyPerLabel(confusion.matrix.two, dataset.two)  
  
## [1] 80.76241 98.98876
```

Se obtuvo una eficacia similar en la detección de ataques que en la evaluación con cinco clusters. La verdadera mejora vino en la eficacia a la hora de clasificar el **tráfico normal**, donde se obtuvo un 98.99% de acierto. En esta oportunidad, la matriz de confusión ya es binaria, por consecuente se pueden calcular las medidas de rendimiento correspondientes.

```
Sensitivity(confusion.matrix.two) * 100  
  
## [1] 80.76241  
  
Especifity(confusion.matrix.two) * 100  
  
## [1] 98.98876  
  
Precision(confusion.matrix.two) * 100  
  
## [1] 98.5822
```

La **sensitividad** nos dice que el modelo fue muy bueno clasificando el **tráfico normal**, por otra parte clasificando los **ataques** es menos efectivo. La medida de **sensitividad** con respecto al modelo con cinco clusters se vio incrementada en 4%, por otra parte, la **especificidad** y la **precisión** es bastante parecida en ambos modelos.

## Conclusión

En general ambos modelos tienen altos valores de eficacia, sin embargo, el modelo con dos clusters obtuvo mejores resultados con respecto a la **eficacia** y **sensitividad** de manera notoria, y algunas mejoras simples en las medidas de **especificidad** y **precisión**. Adicionalmente, se observó que la cantidad de falsos positivos fue reducida en el modelo con dos clusters. Finalmente, el algoritmo **Codo de Jambu** es un buen método para la preselección de clusters a la hora de aplicar **K-Medias**.

## Redes Neuronales

En esta sección se describirán las actividades realizadas para el entrenamiento y evaluación de redes neuronales en el ámbito de detección de intrusos en redes de computadoras. Este sección de subdivide en dos grandes partes **Entrenamiento del modelo** y **Evaluación del modelo**. Esto debido a que los pasos y observaciones serán realizadas de manera individual.

### Entrenamiento del modelo

Para el entrenamiento de la red neuronal se propone una arquitectura con cuarenta neuronas en la **capa de entrada**, una **capa intermedia** con veinte neuronas, y cinco neuronas para la **capa de salida**. La razón para la selección de la arquitectura descrita previamente corresponde a que inicialmente se tienen cuarenta **variables predictoras** que corresponden a la capa de entrada; por otra parte, se tienen cinco clases objetivo que corresponden a las cinco neuronas de la **capa de salida**. El número de capas intermedias y el número de neuronas por capas que debe poseer una red neuronal no está normado en ningún lugar, sólo existen recomendaciones hechas por expertos. Andrew Ng profesor de la *Universidad de Stanford*, menciona en uno de sus cursos de aprendizaje automático en **Coursera** que un modelo con una capa intermedia es suficiente para resolver una gran cantidad de problemas, adicionalmente comenta que en caso de querer utilizar una segunda capa intermedia, es **recomendable** que ambas capas posean igual cantidad de neuronas.

En este modelo se utilizaron 20 neuronas debido a que al haber una gran cantidad de neuronas de entrada, entonces la mitad de las neuronas de entrada parece suficiente. El paquete utilizado para la implementación de redes neuronales se llama **nnet**. Se probó otro paquete llamado **neuralnet**, a diferencia de **nnet**, **neuralnet** permite crear modelos con múltiples capas intermedias, pero es mucho más lento y para las tareas de clasificación hace el proceso más engorroso. Por otra parte, **nnet** a pesar de que sólo permite hacer uso de una capa intermedia, es mucho más rápido para el entrenamiento y el proceso de preparación de los datos para ser pasados como parámetros es más directo.

Se empezará por establecer el ambiente de trabajo eliminando variables parciales, cargando el archivo de funciones y la vista minable del conjunto de entrenamiento.

```
rm(list = ls())
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv",
                           sep = ",", header = TRUE)
source("../source/functions/functions.R")
```

Es importante mencionar que en esta sección se hará el entrenamiento y la evaluación del modelo haciendo uso únicamente del **conjunto de entrenamiento** haciendo uso de la técnica de validación de modelos llamada **validación cruzada de 10 conjuntos**.

Cómo se mencionó previamente el paquete utilizado es **nnet**, a continuación se procederá a instalarlo y a cargarlo en el ambiente de trabajo.

```
install.packages("nnet")
library("nnet")
```

Una vez que tenemos nuestro ambiente de trabajo preparado se eliminarán aquellas etiquetas del conjunto de datos que no van a ser utilizadas a lo largo del proceso de entrenamiento del modelo. Este nivel posee cinco clases objetivos que son **DoS**, **normal**, **Probing**, **R2L** y **U2R**, esto con la finalidad de que la salida para el especialista sea más entendible y pueda identificar la falla de seguridad acontándola dentro de estas cuatro clase de ataques. Dicho esto eliminaremos el resto de las etiquetas.

```
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL
```

Es obligatorio para el uso de las redes neuronales que todas las variables predictoras sean de tipo **numérico**. Por tal motivo, las columnas serán transformadas a tipo numérico. Adicionalmente, como haremos tareas de clasificación, se establecerá la columna objetivo como tipo **factor**.

```
for (i in 1 : (ncol(dataset) -1) )
  dataset[,i] = as.numeric(dataset[,i])

dataset[,ncol(dataset)] = as.factor(dataset[,ncol(dataset)])
```

Para reducir el tiempo de entrenamiento y mejorar la precisión de las predicciones es buena práctica escalar el conjunto de datos a valores que posean *media* cero (0) y *desviación estándar* uno (1).

```
dataset = ScaleSet(dataset)
```

La estrategia adoptada para la evaluación del modelo será la utilización de **validación cruzada de 10 conjuntos**. La función **CVSet** toma un conjunto de datos y establece 10 divisiones de igual longitud del conjunto de datos y las devuelve en una **lista de dataframes**.

```
cv.sets = CVSet(dataset, k = 10, seed = 22)
length(cv.sets)

## [1] 10
```

Se observa que la longitud de la lista es de diez posiciones, esto debido a que en cada posición se posee un dataframe que corresponde a un subconjunto del conjunto de datos original. Todos los registros entre los diferentes dataframes son diferentes debido a que el muestreo se hizo sin reemplazo. Para seguir con las tareas se procederá a inicializar algunas variables.

```
results = vector(mode = "numeric", length = 10)
list.results = list(0, 0, 0, 0)
names(list.results) = c("results", "best_model", "best_testing_set", "best_predictions")
best.accuracy = 0
```

El proceso de entrenamiento y de validación del modelo es bastante largo y por esto se almacenarán en una lista los **resultados** correspondientes a la eficacia de cada modelo en cada iteración, el **mejor modelo**, el **conjunto de datos de prueba** que originó la predicción, y el mejor conjunto de **predicciones**. De esta manera la lista puede ser guardada como un objeto y ser exportada a un archivo que posteriormente puede cargarse, y no es necesario esperar a la realización de este paso cada vez que se deseen analizar los resultados. El siguiente fragmento de código es el encargado de realizar las tareas mencionadas con anterioridad.

```
for (i in 1:10)
{
  #Extrayendo el conjunto de datos
  testingset = as.data.frame(cv.sets[[i]])
  trainingset = cv.sets
  trainingset[[i]] = NULL
  trainingset = do.call(rbind, trainingset)

  #Entrenamiento de la red neuronal
  model = nnet(Label ~ .,
               data = trainingset,
```

```

    size = 20,
    maxit = 100)

#Realizando las predicciones
predictions = predict(model, testingset[, 1:(ncol(testingset)-1)], type = "class")

#Calculando la tasa de aciertos
accuracy = mean(testingset[, ncol(testingset)] == predictions)

#Almacenando el resultado
results[i] = accuracy

#Almacenando el mejor resultado
if(best.accuracy < accuracy)
{
  list.results$best_model = model
  list.results$best_testing_set = testingset
  list.results$best_predictions = predictions
  best.accuracy = accuracy
}
}

```

Una vez que se culmina el proceso de entrenamiento, se almacenan en la lista los resultados parciales de cada iteración y se exporta el modelo.

```

list.results$results = results
saveRDS(list.results, "../source/normal_model/NN/Tests/list_results.rds")

```

### Evaluación del modelo

En esta sección se hará la evaluación de los resultados obtenidos en la sección anterior, adicionalmente se tomará el mejor modelo y las mejores predicciones obtenidas para agregarle el segundo nivel de clasificación correspondiente al algoritmo **K-Medias**.

Se empezará por establecer el ambiente de trabajo eliminando variables parciales, cargando el paquete **nnet**, cargando el archivo de funciones y la lista con información exportada previamente.

```

rm(list = ls())
library("nnet")
source("../source/functions/functions.R")
list.results = readRDS("../source/normal_model/NN/Tests/list_results.rds")

```

A continuación se visualizará la eficacia obtenida de la eficacia del proceso de validación cruzada y se calculará la *media* de los resultados obtenidos.

```

list.results$results

## [1] 0.9952370 0.9946194 0.9945114 0.9949907 0.9960073 0.9954289 0.9952203
## [8] 0.9941909 0.9957588 0.9956967

```

```
mean(list.results$results) * 100
```

```
## [1] 99.51661
```

La **media** de aciertos es de 99.52%. Esta **tasa de aciertos** es bastante alta, lo que demuestra que las redes neuronales pueden tener un buen desempeño en la tarea de detección de intrusos en redes de computadoras. A continuación se creará una matriz de confusión del mejor modelo obtenido en el proceso para visualizar gráficamente el desempeño del algoritmo.

```
confusion.matrix = table(Real = list.results$best_testing_set[,ncol(list.results$best_testing_set)],  
                           Prediction = list.results$best_predictions)  
confusion.matrix
```

```
##          Prediction  
## Real      DoS normal Probing R2L U2R  
## DoS      3036     3      0    1    0  
## normal     4    4417     6    5    1  
## Probing    1      6    711    0    0  
## R2L        0      2      1   67    1  
## U2R        1      1      0    0    1
```

Se observa una matriz de confusión bastante ordenada con pocos registros fuera de la diagonal, es decir, con pocos fallos de clasificación. Ahora se calculará la tasa de acierto y de error.

```
accuracy = mean(list.results$best_testing_set[,ncol(list.results$best_testing_set)] ==  
                  list.results$best_predictions)  
  
accuracy * 100
```

```
## [1] 99.60073
```

```
ErrorRate(accuracy) * 100
```

```
## [1] 0.399274
```

La mejor **tasa de aciertos** fue de 99.6%. Una muy buena tasa de aciertos proporcionada por el modelo de red neuronal. Ahora veamos la eficacia del modelo por etiqueta. Recordemos que la salida corresponde a un vector numérico donde el orden es el siguiente: **DoS, normal, Probing, R2L, U2R**, es decir, el orden alfabético de las etiquetas.

```
AccuracyPerLabel(confusion.matrix, list.results$best_testing_set)
```

```
## [1] 99.86842 99.63907 99.02507 94.36620 33.33333
```

Para las clases **DoS, normal, Probing** y **R2L** la **tasa de aciertos** está por encima del 99.3%, mientras que para la clase **U2R** es de solo el 33.3%. Lo último se debe a la poca cantidad de ejemplos para entrenamiento proporcionados que hace que el algoritmo no pueda generalizar de la manera adecuada, sin embargo, se espera que la eficacia para esta clase incremente conforme se agreguen mayor cantidad de ejemplos para el entrenamiento.

Para poder calcular las medidas de rendimiento binarias correspondientes a la **sensibilidad, especificidad, precisión** y la graficación de la curva **ROC** es necesario llevar la matriz de confusión de cinco clases a una matriz binaria, es decir, con clases **Attack** y **normal**.

```

attack.normal.confusion.matrix = AttackNormalConfusionMatrix(list.results$best_testing_set,
                                                               list.results$best_predictions)
attack.normal.confusion.matrix

##          Prediction
##  Real      Attack normal
##  Attack     3820      12
##  normal     16     4417

```

De esta manera se observa que sólo existen veintiocho errores en el proceso de clasificación, donde doce pertenecen a **falsos negativos** y dieciséis corresponden a **falsos positivos**. Es importante resaltar que el modelo está realizado para que la clase objetivo sea la detección de ataques, esto dicho para la correcta interpretación de la matriz de confusión. Ahora que se tiene la matriz de confusión binaria es posible calcular las medidas de rendimiento binarias mencionadas con anterioridad.

```
Sensitivity(attack.normal.confusion.matrix) * 100
```

```
## [1] 99.68685
```

```
Especifity(attack.normal.confusion.matrix) * 100
```

```
## [1] 99.63907
```

```
Precision(attack.normal.confusion.matrix) * 100
```

```
## [1] 99.5829
```

En las tres medidas se obtuvo un excelente desempeño, todas estas tuvieron un porcentaje superior a 99.5%. Esto nos indica que el modelo es bueno clasificando cualquier tipo de tráfico de manera correcta, es decir, acierta de forma correcta identificando los **ataques** y el **tráfico normal**.

A continuación se graficará la **curva ROC** que nos dará una perspectiva gráfica con la que el modelo clasifica. En este gráfico se grafica la proporción de aciertos contra la proporción de fallos. La correcta interpretación de este gráfico es la siguiente: medir la certeza con la que algoritmo toma sus decisiones. Esto debido a que las predicciones fueron ordenadas de manera descendente utilizando la probabilidad de predicción de los registros, en el inicio del *eje x* se tienen las predicciones realizadas con mayor probabilidad, y a medida que nos desplazamos hacia la derecha del mismo eje la probabilidad de las predicciones va decrementando. Dicho esto, para la creación de la **curva ROC** se necesita un **vector de probabilidades**, un **vector de predicciones** y un **vector real** que corresponde al correcto nombramiento de un registro.

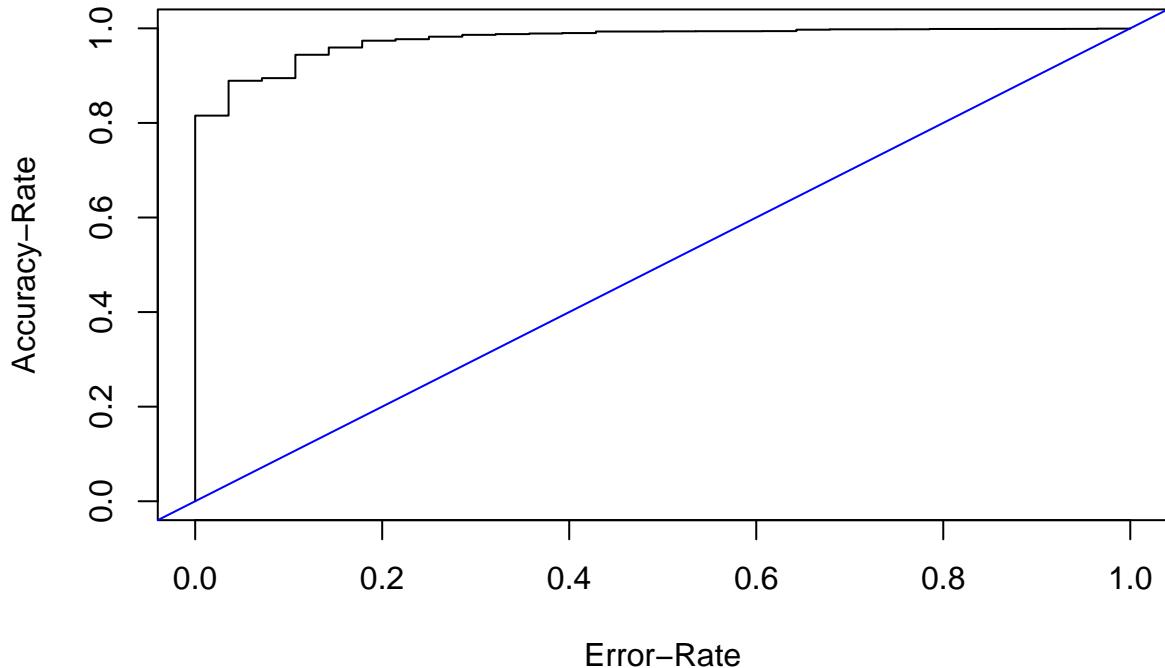
```

probabilities = predict(list.results$best_model,
                        list.results$best_testing_set[, 1:(ncol(list.results$best_testing_set)-1)])
roc.data = DataROC(list.results$best_testing_set, probabilities,
                    list.results$best_predictions)

generate_ROC(scores = roc.data$Prob, real = roc.data$Label,
             pred = roc.data$Prediction)

```

## ROC Curve



En la curva se observa que la mayoría de los aciertos son logrados con una alta probabilidad, conforme la probabilidad va decrementando, el modelo empieza a cometer algunos pocos errores. Al final la mayoría de los errores son cometidos por aquellas predicciones realizadas con baja probabilidad.

### Segundo nivel de clasificación (K-Medias)

A continuación se añadirá el segundo nivel de clasificación que corresponde al uso de **K-Medias** para tomar todos aquellos registros clasificados como **normal** para tratar de corregir los **falsos negativos** producidos por la red neuronal. El algoritmo de K-Medias será implementado con dos clusters debido a que en la sección de **K-Medias** se ilustra que con dos clusters la varianza acumulada es la más adecuada, adicionalmente se probó que con dos clusters se obtuvieron mejores resultados que con cinco.

```
kmeans.set = list.results$best_testing_set[list.results$best_predictions == "normal",]  
kmeans.set[,ncol(kmeans.set)] = as.character(kmeans.set[,ncol(kmeans.set)])  
kmeans.set[kmeans.set[,ncol(kmeans.set)] != "normal",ncol(kmeans.set)] = "Attack"  
SumLabels(kmeans.set, ncol(kmeans.set))  
  
## [1] 12 4417
```

Acá se observa cómo fueron extraídos los 12 registros que no fueron correctamente clasificados, y los otros 4417 registros que si fueron correctamente clasificados como **normal**. El objetivo es clasificar la mayor cantidad de esos 12 registros que son ataques como ataques.

En la sección de **K-Medias** se mencionó que utilizando dos centroides en varias iteraciones se obtuvo el mismo resultado, también se mencionó que esto fue debido a que el algoritmo convergió todas esas veces al mismo **mínimo local**. K-Medias es un algoritmo en el que la preselección de los centroides se hace de manera aleatoria, y es posible obtener diferentes resultados si se hacen múltiples corridas del algoritmo. Por lo anterior, precalcularemos los centroides ejecutando el algoritmo de K-Medias 100 veces y luego promediaremos la posición de los centroides finales. De esta manera, tendremos mejor posicionados los centroides del mejor mínimo local y podremos obtener mejores resultados.

```

matrix.centers = FindCentersKmeans(set = kmeans.set, clusters = 2,
                                    iterations = 100, iter.max = 100)

#Promediando los centroides
matrix.centers = matrix.centers/100
kmeans.model = kmeans(kmeans.set[,1:(ncol(kmeans.set)-1)], centers = matrix.centers,
                      iter.max = 100)

```

Una vez que el modelo fue entrenado, veamos sus predicciones.

```

predictions = OrderKmeans(kmeans.model)

confusion.matrix.kmeans.model = table(Real = kmeans.set[,ncol(kmeans.set)],
                                       Prediction = predictions)

confusion.matrix.kmeans.model

##          Prediction
## Real      Attack normal
##   Attack      0     12
##   normal      1   4416

```

Se observa que 0 de los 12 ataques fueron detectados, es decir, volvemos a tener 12 **falsos negativos**. Dicho esto, aparentemente el uso de K-Medias en esta ocasión no fue eficaz debido a que el desempeño del modelo quedó intacto, esto da indicios a pensar que esos 12 **falsos negativos** están mezclados dentro de lo que es el tráfico normal y no son notablemente separables. Por otra parte hay un aspecto positivo a destacar que es el hecho de que el uso de K-Medias no deterioró de gran manera el trabajo hecho por el modelo de redes neuronales. Se espera que este comportamiento mejore conforme haya mayor cantidad de falsos negativos luego de pasar el primer nivel de clasificación. A continuación se calculará la tasa de **aciertos** y de **errores** del modelo.

```

accuracy.kmeans.model = mean(predictions == kmeans.set[,ncol(kmeans.set)])
accuracy.kmeans.model*100

```

```

## [1] 99.70648

ErrorRate(accuracy.kmeans.model)*100

```

```

## [1] 0.29352

```

Evidentemente la tasa de aciertos es bastante alta debido a que la gran mayoría del tráfico correspondía a **tráfico normal** y el algoritmo clasificó todos los registros salvo uno como **tráfico normal**. A continuación veamos la eficacia por etiqueta. Las posiciones del vector de salida corresponden a la eficacia de las etiquetas **Attack** y **normal** respectivamente.

```

AccuracyPerLabel(confusion.matrix.kmeans.model, kmeans.set)

```

```

## [1] 0.00000 99.97736

```

Se obtuvo 0% de acierto en la predicción de ataques, esto no es bueno debido a que el objetivo es la detección de los ataques, sin embargo, es bueno que la tasa aciertos en el tráfico normal sea tan alta, ya que esto da indicio de que no se generaron muchos **falsos positivos ni falsos negativos**. Ahora veamos las medidas de **sensitividad, especificidad y precisión**.

```
Sensitivity(confusion.matrix.kmeans.model) * 100
```

```
## [1] 0
```

```
Especifity(confusion.matrix.kmeans.model) * 100
```

```
## [1] 99.97736
```

```
Precision(confusion.matrix.kmeans.model) * 100
```

```
## [1] 0
```

La **especificidad** es bastante alta, esto quiere decir que el algoritmo tiene alta precisión detectando el **tráfico normal**, por otra parte, la **sensitividad y precisión** son 0, lo que nos dice que el algoritmo no clasificó bien ningún ataque.

### Estadísticas totales

En esta sección se unificarán las estadísticas de ambos niveles de los modelos utilizados para evaluar el desempeño conjunto. Se empezará por unificar las dos matrices de confusión, para poder calcular las estadísticas utilizadas con anterioridad.

```
confusion.matrix.two.labels = TwoLevelsCM(attack.normal.confusion.matrix,
                                            confusion.matrix.kmeans.model)
```

```
confusion.matrix.two.labels
```

```
##      [,1] [,2]
## [1,] 3820   12
## [2,]   17 4416
```

Se observa que la matriz de confusión quedó muy parecida a la matriz de confusión del modelo de red neuronal, salvo que ahora hay un **falso positivo** más. Ahora calcularemos las medidas de rendimiento de **tasa de acierto, tasa de error, sensitividad, especificidad y precisión**.

```
accuracy.total = Accuracy(confusion.matrix.two.labels)
accuracy.total * 100
```

```
## [1] 99.64912
```

```
ErrorRate(accuracy.total) * 100
```

```
## [1] 0.3508772
```

```

Sensitivity(confusion.matrix.two.labels) * 100

## [1] 99.68685

Especifity(confusion.matrix.two.labels) * 100

## [1] 99.61651

Precision(confusion.matrix.two.labels) * 100

## [1] 99.55695

```

Las medidas prácticamente iguales a las del modelo de **red neuronal**, esto es porque la aplicación de **K-Medias** no proporcionó ningún aspecto positivo ni negativo significante.

## Conclusiones

Individualmente el modelo de red neuronal posee un excelente rendimiento con bajas tasas de **falsos positivos** y **falsos negativos**. Al combinarse con **K-Medias** no se logró ninguna mejora, tampoco esto deterioró el modelo, aspecto que es positivo. La situación del segundo modelo espera que mejore en escenarios donde el primer nivel tenga una mayor cantidad de **falsos negativos**, donde quizás los elementos queden con divisiones más obvias y detectables por el algoritmo de **K-Medias**.

## Máquinas de Soporte Vectorial

En esta sección se describirán las actividades realizadas para el entrenamiento y evaluación de **máquinas de soporte vectorial** en el ámbito de la detección de intrusos en redes de computadoras. Esta sección se subdivide en dos grandes partes, que son **Entrenamiento del modelo** y **Evaluación del modelo**. Esto debido a que los pasos y observaciones a ambos procesos serán realizadas de manera individual.

### Entrenamiento del modelo

Para el entrenamiento de la **máquina de soporte vectorial** se utilizará el **kernel radial**, esto debido a que Bhavsar y Waghmare expusieron en su publicación *Intrusion Detection System Using Data Mining Technique: Support Vector Machine* que el kernel radial es más preciso y veloz a la hora de entrenar y clasificar que los otros kernels **linear**, **polinomial** o **sigmoid**. Los parámetros para el algoritmo de SVM con kernel radial tiene los parámetros **cost** y **gamma** que deben ser elegidos, sin embargo, en esta sección se utilizarán los parámetros por defecto, ya que el objetivo acá es probar el desempeño general del modelo. Más adelante, en la sección de selección de parámetros, se seleccionarán los mejores parámetros para el modelo. Se utilizará el paquete **e1071** debido a que en la documentación en blogs y tutoriales es el más utilizado y es el que posee más documentación.

Se iniciará la descripción de las actividades realizadas eliminando variables parciales, cargando el archivo de funciones y la vista minable del conjunto de entrenamiento.

```

rm(list = ls())
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv",
sep = ",", header = TRUE)
source("../source/functions/functions.R")

```

Es importante mencionar que en esta sección, al igual que en la de **redes neuronales** se hará el entrenamiento y la evaluación del modelo haciendo uso únicamente del **conjunto de entrenamiento**, y aplicando la técnica de validación de modelos llamada **validación cruzada de 10 conjuntos**.

Cómo se mencionó con anterioridad el paquete a ser utilizado es **e1071**, por consiguiente se procederá a instalarlo y cargarlo en el ambiente de trabajo.

```
install.packages("e1071")
library("e1071")
```

Una vez que se tiene nuestro ambiente de trabajo preparado, se eliminarán aquellas etiquetas del conjunto de datos que no van a ser utilizadas a lo largo del proceso de entrenamiento del modelo. Este nivel del modelo híbrido poseerá cinco clases objetivos que son **DoS**, **normal**, **Probing**, **R2L**, y **U2R**; esto con la finalidad de que la salida para el especialista sea más entendible y pueda identificar las fallas de seguridad acontándolas dentro de estas cuatro clases de ataques. Dicho esto, eliminaremos el resto de las etiquetas.

```
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL
```

Es necesario que todos los tipos de datos de las diferentes columnas sean de tipo numérico para poder entrenar a la máquina de soporte vectorial. Por este motivo, las columnas predictoras serán transformadas a tipo **numérico**. Adicionalmente, como faremos tareas de clasificación, la columna objetivo la transformaremos a tipo **factor**.

```
for (i in 1 : (ncol(dataset) -1) )
  dataset[,i] = as.numeric(dataset[,i])

dataset[,ncol(dataset)] = as.factor(dataset[,ncol(dataset)])
```

Para reducir el tiempo de entrenamiento y mejorar la precisión de las predicciones es buena práctica escalar el conjunto de datos a valores que posean *media* cero (0) y *desviación estándar* uno (1).

```
dataset = ScaleSet(dataset)
```

La estrategia utilizada para la evaluación del modelo será la utilización de **validación cruzada de 10 conjuntos**. La función **CVSet** toma un conjunto de datos y establece 10 divisiones de igual longitud del conjunto de datos y las retorna en una lista de **dataframes**.

```
cv.sets = CVSet(dataset, k = 10, seed = 22)
length(cv.sets)
```

```
## [1] 10
```

Se observa que la longitud de la lista es de diez posiciones, esto debido a que en cada posición se posee un datafrme que corresponde a un subconjunto del conjunto de datos original. Todos los registros entre los dataframes son diferentes debido a que el muestreo se hizo sin reemplazo. Para seguir con las actividades se procederá a inicializar algunas variables.

```

results = vector(mode = "numeric", length = 10)
list.results = list(0, 0, 0, 0)
names(list.results) = c("results", "best_model", "best_testing_set", "best_predictions")
best.accuracy = 0

```

El proceso de entrenamiento y validación del modelo es bastante largo y por eso se almacenarán en una lista los **resultados** correspondientes a la eficacia del modelo en cada iteración, el **mejor modelo**, el **conjunto de datos de prueba**, que originó la predicción, y el mejor conjunto de **predicciones**. De esta manera, la lista puede ser guardada como un objeto y ser exportada a un archivo que posteriormente puede cargarse y no es necesario esperar a la realización de este paso cada vez que se deseen analizar los resultados. El siguiente fragmento de código es el encargado de realizar las tareas mencionadas con anterioridad.

```

for (i in 1:10)
{
  #Extrayendo el conjunto de datos
  testingset = as.data.frame(cv.sets[[i]])
  trainingset = cv.sets
  trainingset[[i]] = NULL
  trainingset = do.call(rbind, trainingset)

  #Entrenamiento de SVM
  model = svm(Label ~ .,
              data = trainingset,
              kernel = "radial",
              scale = FALSE,
              probability = TRUE)

  #Realizando las predicciones
  predictions = predict(model, testingset[, 1:(ncol(testingset)-1)],
                        type = "class")

  #Calculando la tasa de aciertos
  accuracy = mean(testingset[, ncol(testingset)] == predictions)

  #Almacenando el resultado
  results[i] = accuracy

  #Almacenando el mejor resultado
  if(best.accuracy < accuracy)
  {
    list.results$best_model = model
    list.results$best_testing_set = testingset
    list.results$best_predictions = predictions
    best.accuracy = accuracy
  }
}

```

Una vez que se culmina el proceso de entrenamiento y validación, se almacenan en la lista los resultados parciales de cada iteración y se exporta el modelo.

```
list.results$results = results
saveRDS(list.results, "../source/normal_model/SVM/Tests/list_results.rds")
```

## Evaluación del modelo

En esta sección se hará la evaluación de los resultados obtenidos en la sección anterior, adicionalmente se tomará el mejor modelo y las mejores predicciones para agregar el segundo nivel de clasificación correspondiente al algoritmo **K-Medias**.

Se empezará por establecer el ambiente de trabajo eliminando variables parciales, cargando el paquete **e1071**, cargando el archivo de funciones y la lista con la información exportada previamente.

```
rm(list = ls())
library("e1071")
source("../source/functions/functions.R")
list.results = readRDS("../source/normal_model/SVM/Tests/list_results.rds")
```

A continuación se visualizará la eficacia obtenida en cada iteración del proceso de validación cruzada y se calculará la media de acierto.

```
list.results$results

## [1] 0.9919028 0.9909147 0.9915711 0.9923772 0.9937084 0.9920678 0.9914862
## [8] 0.9908714 0.9933616 0.9907787

mean(list.results$results) * 100

## [1] 99.1904
```

La media de aciertos es de 99.19%. Esto demuestra que las máquinas de soporte vectorial pueden tener un muy buen desempeño en la tarea de detectar anomalías conocidas en redes de computadoras. A continuación se creará una matriz de confusión del mejor modelo obtenido en el proceso para visualizar gráficamente el desempeño del algoritmo.

```
confusion.matrix = table(Real = list.results$best_testing_set[,ncol(list.results$best_testing_set)],
                         Prediction = list.results$best_predictions)
confusion.matrix

##          Prediction
## Real      DoS normal Probing R2L U2R
## DoS      3035      5       0     0   0
## normal     4    4407     12     9   1
## Probing    1     13    704     0   0
## R2L        0      5       0    66   0
## U2R        0      2       0     0   1
```

Se puede ver una matriz bastante ordenada con pocos elementos fuera de la diagonal. A continuación se visualizará la tasa de aciertos para el mejor modelo.

```

accuracy = mean(list.results$best_testing_set[,ncol(list.results$best_testing_set)] ==
               list.results$best_predictions)

accuracy * 100

## [1] 99.37084

ErrorRate(accuracy) * 100

## [1] 0.6291591

```

La mejor tasa de aciertos fue de 99.37%. Una tasa de aciertos bastante buena, ahora observemos la eficacia por etiquetas del modelo. Recordemos que la salida es un vector numérico ordenado alfabéticamente por el nombre de las etiquetas; es decir, **DoS**, **normal**, **Probing**, **R2L**, **U2R**.

```

AccuracyPerLabel(confusion.matrix, list.results$best_testing_set)

## [1] 99.83553 99.41349 98.05014 92.95775 33.33333

```

Para las clases **DoS**, **normal**, **Probing** y **R2L**, el desempeño es bastante bueno, en todos los casos por encima del 92.95%. Por otra parte, para la clase **U2R** es de sólo 33.33%, esto es debido a la poca cantidad de registros presentes en el conjunto de datos para entrenar al modelo con esta clase de ataques que hace que el algoritmo no pueda generalizar de buena manera para esta clase de ataques. Se espera que incrementando la cantidad de registros de esta clase de ataques aumente la eficacia para su detección.

Para poder calcular las medidas de rendimiento binarias correspondientes a la **sensibilidad**, **especificidad**, **precisión** y la graficación de la **curva ROC** es necesario llevar a la matriz de cinco clases a una matriz binaria, es decir, con clases **Attack** y **normal**\*\*.

```

attack.normal.confusion.matrix = AttackNormalConfusionMatrix(list.results$best_testing_set,
                                                               list.results$best_predictions)
attack.normal.confusion.matrix

##          Prediction
## Real      Attack normal
##   Attack    3807     25
##   normal     26    4407

```

De esta manera se observa que sólo existen 55 errores en la clasificación, de los cuales 25 pertenecen a **falsos negativos** y 26 a **falsos positivos**. Es importante resaltar que el modelo fue hecho para que la clase objetivo sea la detección de ataques, esta información es importante para la correcta interpretación de la matriz de confusión. Ahora que se tiene la matriz de confusión binaria es posible calcular las medidas de rendimiento mencionadas previamente.

```

Sensitivity(attack.normal.confusion.matrix) * 100

## [1] 99.3476

```

```

Especifity(attack.normal.confusion.matrix) * 100

## [1] 99.41349

Precision(attack.normal.confusion.matrix) * 100

## [1] 99.32168

```

En las tres medidas se obtuvo un excelente desempeño, todas tuvieron un porcentaje superior a 99.32%. Esto nos indica que el modelo es bueno clasificando el tráfico de manera correcta, es decir, acierta de buena manera identificando **ataques** y **tráfico normal**.

A continuación se graficará la **curva ROC** que nos dará una perspectiva gráfica de la certeza de las clasificaciones del modelo. En este gráfico se ilustra la proporción de aciertos contra la proporción de fallos ordenados por la probabilidad de la toma de la decisión del modelo al clasificar cierto registro; por consiguiente, en el inicio del *eje X* se obtendrán aquellas predicciones que fueron hechas con mayor puntaje de certeza; a medida que nos desplazamos hacia la derecha en dicho eje, el puntaje de las certezas va decrementando. Dicho esto, para la creación de la **curva ROC** se necesita de un **vector de probabilidades**, un **vector de predicciones** y un **vector real** que corresponde al correcto nombramiento del registro.

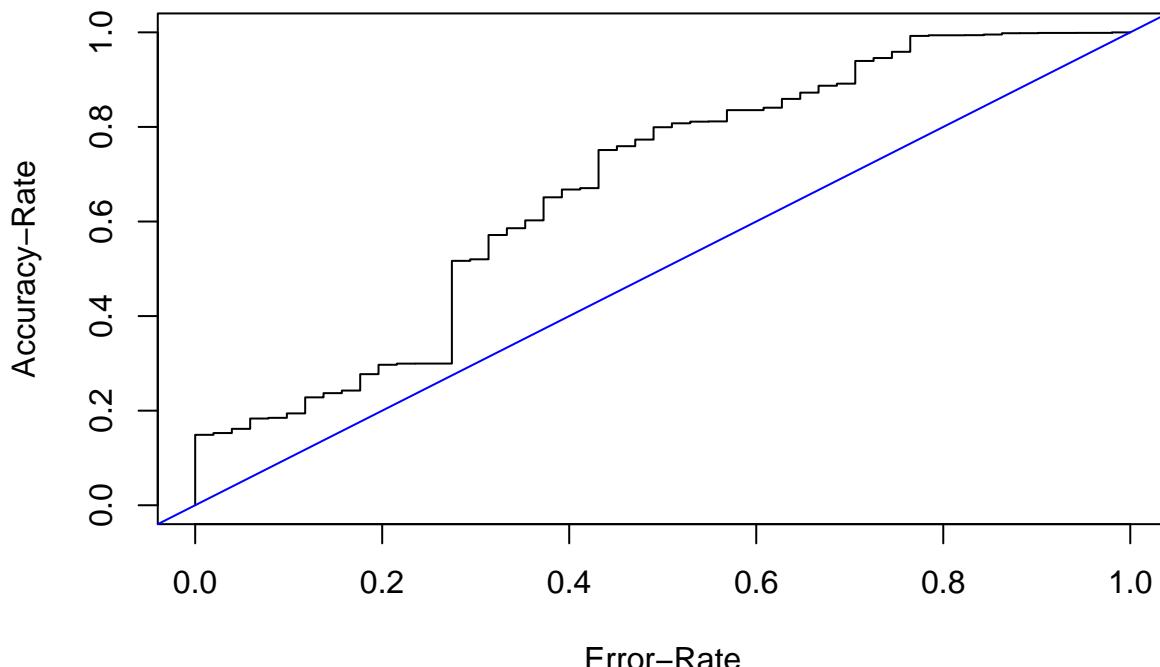
```

probabilities = attr(predict(list.results$best_model,
                             list.results$best_testing_set[, 1:(ncol(list.results$best_testing_set)-1)],
                             probability = TRUE), "probabilities")

roc.data = DataROC(list.results$best_testing_set, probabilities,
                    list.results$best_predictions)
generate_ROC(scores = roc.data$Prob, real = roc.data$Label,
             pred = roc.data$Prediction)

```

**ROC Curve**



En la curva se muestra como el comportamiento es un tanto errático con respecto a la certeza con la que se toman las decisiones, es decir, hay decisiones erroneas que se toman con alta certeza. El mejor rendimiento se alcanza con valores de certeza intermedios que es donde la función se separa más de la línea de identidad.

### Segundo nivel de clasificación (K-Medias)

A continuación se añadirá el segundo nivel de clasificación que corresponde al uso de **K-Medias** para tomar todos aquellos registros clasificados como **normal** para tratar de corregir los falsos negativos producidos por la **máquina de soporte vectorial**. El algoritmo de K-Medias será implementado con dos clusters debido a que en la *sección de K-Medias* se ilustra que con dos clusters la varianza acu,ulada es óptima, adicionalmente se probó que con dos clusters se obtuvieron mejores resultados que con cinco.

```
kmeans.set = list.results$best_testing_set[list.results$best_predictions == "normal",]
kmeans.set[,ncol(kmeans.set)] = as.character(kmeans.set[,ncol(kmeans.set)])
kmeans.set[kmeans.set[,ncol(kmeans.set)] != "normal",ncol(kmeans.set)] = "Attack"
SumLabels(kmeans.set, ncol(kmeans.set))
```

```
## [1] 25 4407
```

Acá se observa como fueron extraídos los 25 registros que no fueron correctamente clasificados y los 4407 registros que si fueron correctamente clasificados como **tráfico normal**. El objetivo es clasificar la mayor cantidad de esos registros que son ataques como ataques.

En la sección de **k-Medias** se mencionó que utilizando dos centroides en varias iteraciones se obtuvo el mismo resultado, también se mencionó que esto fue debido a que el algoritmo convergió todas esas veces al mismo **mínimo local**. k-Medias es un algoritmo en el que la preselección de los centroides se hace de manera aleatoria, por esta razón, es posible obtener diferentes resultados si se hacen multiples corridas del algoritmo. Por lo anterior, se precalcularán los centroides ejecutando el algoritmo de K-Medias 100 veces y luego se promediará la posición de los centroides finales. De esta manera, se tendrán mejor posicionados los centroides desde el inicio permitiéndonos acercarnos al mínimo local y obtener mejores resultados.

```
matrix.centers = FindCentersKmeans(set = kmeans.set, clusters = 2,
                                    iterations = 100, iter.max = 100)

#Promediando los centroides
matrix.centers = matrix.centers/100
kmeans.model = kmeans(kmeans.set[,1:(ncol(kmeans.set)-1)], centers = matrix.centers,
                      iter.max = 100)
```

Una vez que el modelo fue entrenado, veamos sus predicciones.

```
predictions = OrderKmeans(kmeans.model)
confusion.matrix.kmeans.model = table(Real = kmeans.set[,ncol(kmeans.set)],
                                         Prediction = predictions)
confusion.matrix.kmeans.model

##          Prediction
## Real      Attack normal
##   Attack      0      25
##   normal      1    4406
```

Se observa que 0 de los 25 ataques fueron detectados, es decir, volvemos a tener 25 **falsos negativos**. Dicho esto, aparentemente el uso de K-Medias en esta ocasión no fue eficaz debido a que el desempeño del modelo

quedó intacto, esto da indicios a pensar que esos 25 **falsos negativos** están mezclados dentro lo que es el tráfico normal y no son notablemente separables. Por otra parte hay un aspecto positivo a destacar que es el hecho de que el uso de K-Medias no deterioró de gran manera el trabajo hecho por el modelo de máquina de soporte vectorial. Se espera que este comportamiento mejore conforme haya mayor cantidad de **falsos negativos** luego de pasar el primer nivel de clasificación. A continuación se calcularán las tasas de acierto y de error del modelo.

```
accuracy.kmeans.model = mean(predictions == kmeans.set[,ncol(kmeans.set)])
accuracy.kmeans.model*100
```

```
## [1] 99.41336
```

```
ErrorRate(accuracy.kmeans.model)*100
```

```
## [1] 0.5866426
```

Evidentemente la tasa de aciertos es bastante alta debido a que la gran mayoría del tráfico correspondía a **tráfico normal** y el algoritmo clasificó todos los registros salvo uno como **tráfico normal**. A continuación veamos la eficacia por etiqueta. Las posiciones del vector de salida corresponden a las clases **Attack** y **normal** respectivamente.

```
AccuracyPerLabel(confusion.matrix.kmeans.model, kmeans.set)
```

```
## [1] 0.00000 99.97731
```

Se obtuvo 0% de acierto en la predicción de ataques, esto no es bueno debido a que el objetivo es la detección de ataques; sin embargo, es bueno que la tasa de aciertos para **tráfico normal** sea tan alta, ya que esto refleja que no se generaron muchos **falsos positivos** ni **falsos negativos**. Ahora veamos las medidas de **sensitividad, especificidad y precisión**.

```
Sensitivity(confusion.matrix.kmeans.model) * 100
```

```
## [1] 0
```

```
Especifity(confusion.matrix.kmeans.model) * 100
```

```
## [1] 99.97731
```

```
Precision(confusion.matrix.kmeans.model) * 100
```

```
## [1] 0
```

La especificidad es bastante alta, esto quiere decir que el modelo es excelente clasificando el **tráfico normal**, por otra parte, la sensitividad y la especificidad son cero; en consecuencia, el modelo tiene pobre desempeño clasificando los **ataques**.

## Estadísticas totales

A continuación se unificarán las estadísticas de ambos niveles de los modelos utilizados para evaluar el desempeño conjunto. Se empezará por unificar las dos matrices de confusión para poder calcular las estadísticas utilizadas con anterioridad.

```

confusion.matrix.two.labels = TwoLevelsCM(attack.normal.confusion.matrix,
                                         confusion.matrix.kmeans.model)
confusion.matrix.two.labels

##      [,1] [,2]
## [1,] 3807   25
## [2,]    27 4406

```

Se observa que la matriz de confusión quedó muy parecida a la matriz de confusión del modelo de máquina de soporte vectorial, salvo que ahora hay un **falso positivo** más. Ahora calcularemos las medidas de rendimiento de **tasa de acierto**, **tasa de error**, **sensitividad**, **especificidad** y **precisión**.

```

accuracy.total = Accuracy(confusion.matrix.two.labels)
accuracy.total * 100

```

```
## [1] 99.37084
```

```
ErrorRate(accuracy.total) * 100
```

```
## [1] 0.6291591
```

```
Sensitivity(confusion.matrix.two.labels) * 100
```

```
## [1] 99.3476
```

```
Especificity(confusion.matrix.two.labels) * 100
```

```
## [1] 99.39093
```

```
Precision(confusion.matrix.two.labels) * 100
```

```
## [1] 99.29577
```

Las medidas quedaron prácticamente invariantes con respecto al modelo de **máquina de soporte vectorial**, esto debido a que la aplicación de **K-Medias** fue irrelevante.

## Conclusiones

Individualmente el modelo de **SVM** posee un muy buen desempeño con bajas tasas de **falsos positivos** y **falsos negativos**. Con respecto a la **Curva ROC**, se observa que el modelo comete errores tomando decisiones con un elevado valor de certeza, situación que deteriora un poco su rendimiento.

Al combinarse el modelo de **SVM** con **K-Medias** no se logró absolutamente nada, no se mejoró el proceso de detección de intrusos. Cómo aspecto favorable se puede rescatar el hecho de que no se deterioró el rendimiento del primer nivel del modelo híbrido. Por último, la situación con la inclusión de **K-Medias** se espera que mejore conforme se cometan más fallos de tipo **falsos negativos** por parte del primer nivel.

## Conclusiones generales

Los modelos de **red neuronal** y **máquina de soporte vectorial** de manera individual funcionan de gran manera, con bajas tasas de **falsos positivos**, **falsos negativos**, y una gran tasa de acierto. Comparándolos entre ellos, el modelo de **red neuronal** tiene un mejor desempeño individualmente en cada una de las clases de ataques presentes en el conjunto de datos. Adicionalmente, las decisiones que toma con alta certeza suelen ser acertadas, esta característica se puede observar en la **curva ROC**.

La inclusión del segundo nivel de **K-Medias** se comportó en ambos casos de igual manera. Esta no aportó absolutamente nada a la detección de ataques, sin embargo, un aspecto positivo fue que esta no afectó de gran manera el desempeño del primer nivel. Por lo que se espera que con mayor cantidad de ataques no detectados por el primer nivel **K-Medias** pueda tener un mejor desempeño.

## Análisis sobre el conjunto de prueba

En esta sección se listarán las actividades concernientes al entrenamiento y evaluación del modelo haciendo uso del conjunto de datos para la fase de entrenamiento y haciendo uso del conjunto de prueba para la fase de evaluación. Hasta este punto en el documento se ha probado que ante ataques conocidos los modelos de **SVM** y **NN** tienen un muy buen desempeño a la hora de clasificar ataques y el segundo nivel de **K-Medias** no parece ser útil. En esta oportunidad el conjunto de prueba tendrá ataques no incluidos en el conjunto de entrenamiento, situación que permitirá medir la capacidad de generalización de los modelos de **SVM** y **NN**. Adicionalmente, se espera que si hay mayor cantidad de **falsos negativos** en el primer nivel, entonces **K-Medias** pueda ser de utilidad. Se iniciará haciendo el análisis sobre el uso de las redes neuronales.

### Redes Neuronales

En esta sección se describirán las actividades realizadas para el entrenamiento y evaluación de las redes neuronales en el ámbito de la detección de intrusos en redes de computadoras. Esta sección se subdivide en dos grandes partes concernientes al **Entrenamiento del modelo** y **Evaluación del modelo**. Esto debido a que los pasos y observaciones se harán de manera individual en cada fase.

### Entrenamiento del modelo

Se aplicará el mismo criterio que se propuso en la sección de *análisis sobre el conjunto de entrenamiento*; es decir, se usará una arquitectura con 40 neuronas de entrada, una capa intermedia de 20 neuronas y una capa de salida de 5 neuronas. Se usará un modelo de 5 clases donde 4 corresponden a las etiquetas de los ataques y 1 a la etiqueta del tráfico normal. De igual manera, se hará uso del paquete **nnet**. La única diferencia es que ahora se hará uso del conjunto total del conjunto de entrenamiento para el entrenamiento del modelo y se hará la evaluación del modelo haciendo uso del conjunto de datos de prueba, en esta ocasión no se hará uso de **validación cruzada de 10 conjuntos** como técnica de validación.

Se empezará establecer el ambiente de trabajo eliminando variables parciales, cargando el archivo de funciones y la vista minable del conjunto de entrenamiento.

```
rm(list = ls())
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv", sep = ",", header = TRUE)
source("../source/functions/functions.R")
```

El paquete utilizado para el entrenamiento de las redes neuronales es **nnet**, a continuación el paquete se cragará.

```
library("nnet")
```

Una vez que tenemos nuestro ambiente de trabajo preparado se eliminarán aquellas etiquetas del conjunto de datos que no van a ser utilizadas a lo largo del proceso de entrenamiento del modelo. El primer nivel de detección del modelo híbrido posee cinco clases objetivo que son **DoS**, **normal**, **Probing**, **R2L** y **U2R**; esto con la finalidad de que la salida para el especialista sea más entendible y pueda identificar la(s) falla(s) de seguridad acotándola dentro de estas cuatro clases de ataques. Dicho esto eliminaremos el resto de las etiquetas.

```
dataset = dataset.training  
dataset$Label_Normal_TypeAttack = NULL  
dataset$Label_Num_Classifiers = NULL  
dataset$Label_Normal_or_Attack = NULL
```

Es obligatorio que para el uso de las redes neuronales todas las variables predictoras sean de tipo numérico. Por lo tanto, se transformarán cada una de estas a tipo numérico y la columna objetivo se transformará en tipo factor debido a que se realizarán labores de clasificación.

```
for (i in 1 : (ncol(dataset.training) -1) )  
  dataset.training[,i] = as.numeric(dataset.training[,i])  
  
dataset.training[,ncol(dataset.training)] = as.factor(dataset.training[,ncol(dataset.training)])
```

Para acelerar el tiempo de entrenamiento y tener un modelo más preciso es buena práctica escalar el conjunto de datos a rangos similares. En este caso, todas las columnas predictoras tendrán *media* cero (0) y *desviación estándar* uno (1).

```
dataset.training = ScaleSet(dataset.training)
```

Ya se tienen el conjunto de datos listo y el ambiente de trabajo preparado, a continuación se iniciará el proceso de entrenamiento. De igual manera que se realizó en la sección *análisis sobre el conjunto de prueba* el modelo creado será guardado en un objeto debido a que el proceso de entrenamiento es largo y es tedioso tener que esperar a su entrenamiento cada vez que se quiera analizar el modelo. Adicionalmente, en esta oportunidad se calculará el tiempo que tarda el modelo entrenándose. Esto, para poder comparar el tiempo contra la **máquina de soporte vectorial** y luego contra el tiempo de entrenamiento luego de hacer la selección de características y selección de parámetros.

```
start.time = Sys.time()  
set.seed(22)  
  
model = nnet(Label ~ .,  
             data = dataset.training,  
             size = 20,  
             maxit = 100)  
  
total.time = Sys.time() - start.time
```

Por último, el tiempo y el modelo creado se guardan en una lista y se exportan como un objeto para su posterior uso.

```

list.results = list(total.time, model)
saveRDS(list.results, file = "../source/normal_model/NN/Real_Model/list_results.rds")

```

## Evaluación del modelo

En esta sección se hará la evaluación de los resultados obtenidos en la sección anterior, adicionalmente se tomará el mejor modelo y las mejores predicciones obtenidas para agregarle el segundo nivel de clasificación correspondiente al algoritmo K-Medias.

Se empezará por establecer el ambiente de trabajo eliminando variables parciales, cargando el paquete **nnet**, cargando el archivo de funciones, la lista con información exportada previamente y el conjunto de datos de prueba.

```

rm(list = ls())
library("nnet")
source("../source/functions/functions.R")
results = readRDS("../source/normal_model/NN/Real_Model/list_results.rds")
testing.set = read.csv("../dataset/NSLKDD_Testing_New.csv",
                      sep = ",", header = TRUE)

```

Se empezará por eliminar las etiquetas innecesarias, transformar las variables predictoras a tipo **numérico** y la columna objetivo a tipo **factor**, y escalar las variables predictoras dentro de la misma *media* y *desviación estándar*.

```

#Eliminando etiquetas
testing.set$Label_Normal_TypeAttack = NULL
testing.set$Label_Num_Classifiers = NULL
testing.set$Label_Normal_or_Attack = NULL

#Cambiando el tipo de dato
for (i in 1 : (ncol(testing.set) -1) )
  testing.set[,i] = as.numeric(testing.set[,i])

testing.set[,ncol(testing.set)] = as.factor(testing.set[,ncol(testing.set)])

#Escalando las variables predictoras
testing.set = ScaleSet(testing.set)

```

Hasta este punto ya se tienen listos el ambiente de trabajo, conjunto de datos y la lista de resultados de la sección anterior. A continuación se extraerá el modelo y el tiempo de entrenamiento del modelo y se visualizará el tiempo correspondiente al entrenamiento del modelo.

```

training.time = results[[1]]
model = results[[2]]
training.time

## Time difference of 4.444731 mins

```

A partir de este punto se empezará con el análisis del modelo. Todos los pasos que involucrados con el tiempo de entrenamiento y predicción serán cronometrados y al final serán sumados para tener una perspectiva del tiempo necesario para cada fase. Se iniciará con el cálculo de las predicciones.

```

start.time.predictions = Sys.time()
predictions = predict(model, testing.set[, 1:(ncol(testing.set)-1)], type = "class")
total.time.predictions = Sys.time() - start.time.predictions
total.time.predictions

```

```
## Time difference of 0.2173347 secs
```

A continuación, se creará una matriz de confusión que nos ayude a ver gráficamente el desempeño del modelo durante el proceso de clasificación.

```
confusion.matrix = table(Real = testing.set[,ncol(testing.set)],
                         Prediction = predictions)
```

```
confusion.matrix
```

		Prediction				
##	Real	DoS	normal	Probing	R2L	U2R
##	DoS	5942	1428	87	1	0
##	normal	114	9338	225	32	2
##	Probing	262	477	1679	3	0
##	R2L	14	2361	25	354	0
##	U2R	5	154	26	11	4

Si se compara con la matriz de confusión del modelo en la sección de *análisis sobre el conjunto de entrenamiento* se observa una matriz de confusión mucho más desordenada. Sin embargo, a simple vista se observa que la diagonal acumula la mayoría de los registros, adicionalmente se observa que existen más **falsos negativos** que **falsos positivos**, es decir, hubo más errores en los que se clasificó **tráfico normal** como **ataques** que ataques que se clasificaron como **tráfico normal**. A continuación veamos la **tasa de aciertos** y la **tasa de errores**.

```
accuracy = mean(testing.set[,ncol(testing.set)] == predictions)
accuracy * 100
```

```
## [1] 76.81423
```

```
ErrorRate(accuracy) * 100
```

```
## [1] 23.18577
```

Ya no se tiene un desempeño tan alto como se tuvo en el *análisis sobre el conjunto de entrenamiento* y es entendible, debido a que en el conjunto de prueba hay clases ataques que no estuvieron presentes en conjunto de entrenamiento. Sin embargo, una tasa de aciertos de 76.81% es bastante alta, y se espera que la inclusión de **K-Medias** incremente la tasa de aciertos. Ahora veamos la precisión por etiquetas, recordemos que la salida corresponde a un vector que corresponde al siguiente orden: **DoS**, **normal**, **Probing**, **R2L** y **U2R**.

```
AccuracyPerLabel(confusion.matrix, testing.set)
```

```
## [1] 79.67283 96.15899 69.35151 12.85403 2.00000
```

Para las etiquetas de **DoS**, **normal** y **Probing** el rendimiento es bastante bueno, en especial para **DoS** y **normal**. Sin embargo, para **R2L** y **U2R** es bastante pobre. Esto puede deberse a la poca cantidad de registros usados para el entrenamiento en ambos casos, en particular para la clase **U2R**.

A continuación crearemos una matriz de confusión binaria para poder calcular las medidas de rendimiento binarias correspondientes a **sensitividad**, **especificidad**, **precisión** y la graficación de la **curva ROC**.

```
attack.normal.confusion.matrix = AttackNormalConfusionMatrix(testing.set, predictions)
attack.normal.confusion.matrix
```

```
##          Prediction
## Real      Attack normal
##   Attack    8413    4420
##   normal     373    9338
```

Se nota una baja cantidad de **falsos positivos** y una alta cantidad de **falsos negativos** y una alta tasa de aciertos con respecto a la clasificación de los registros en la diagonal. Ahora que hay mayor cantidad de **falsos negativos** el algoritmo de **K-Medias** puede aportar más al tema de la clasificación. Ahora veamos las medidas de rendimiento binarias mencionadas con anterioridad.

```
Sensitivity(attack.normal.confusion.matrix) * 100
```

```
## [1] 65.55755
```

```
Especifity(attack.normal.confusion.matrix) * 100
```

```
## [1] 96.15899
```

```
Precision(attack.normal.confusion.matrix) * 100
```

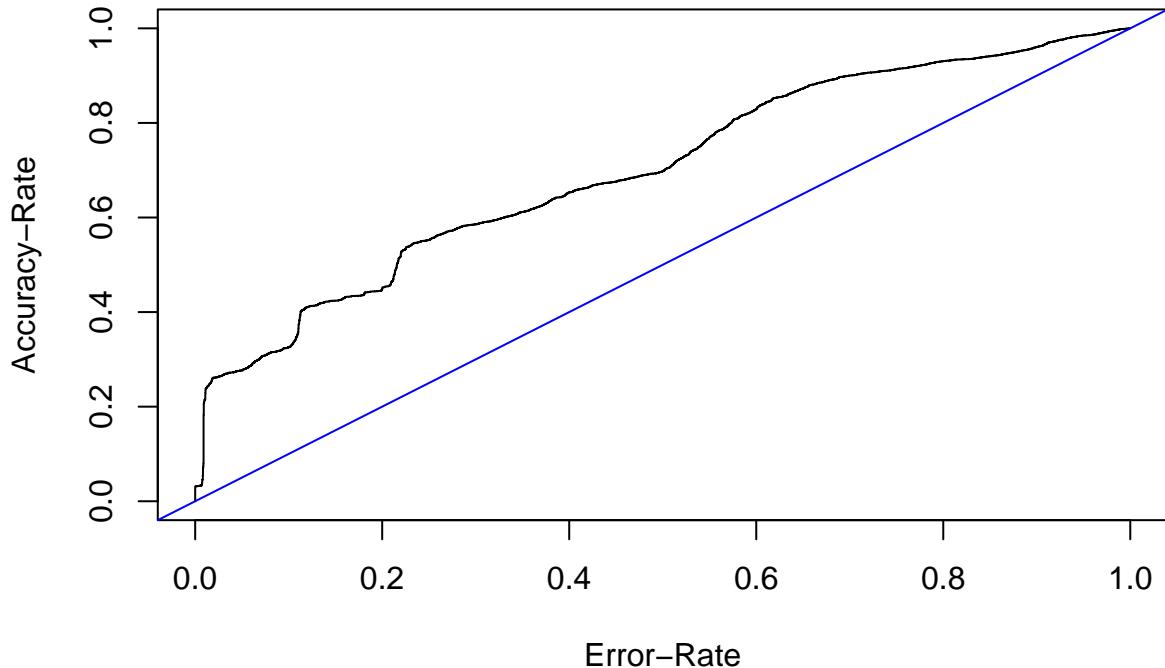
```
## [1] 95.75461
```

La **sensitividad** nos dice que el 65.56% de los **ataques** presentes en el conjunto de datos fueron detectados. Por otra parte, la **especificidad** nos dice que el 95.16% de los registros pertenecientes al **tráfico normal** fueron detectados. Por último, la **precisión** nos dice que el 95.75% de las clasificaciones de **ataques** fueron correctas.

En general el modelo tiene un buen desempeño en la detección de **tráfico normal** y la mayoría de las predicciones que hace cómo **ataques** son verdaderas, sin embargo, el gran problema son los **falsos negativos** que se espera que con la utilización de **k-Medias** la tasa de aciertos pueda mejorar. A continuación graficaremos la **curva ROC**.

```
probabilities = predict(model, testing.set[, 1:(ncol(testing.set)-1)])
roc.data = DataROC(testing.set, probabilities, predictions)
generate_ROC(roc.data$Prob, roc.data$Label, roc.data$Prediction)
```

## ROC Curve



En comparación con la sección de *análisis sobre el conjunto de entrenamiento*, se observa un desempeño notablemente inferior, en esta ocasión la curva no tiene tanta distancia de separación de la función de identidad y se puede observar cómo ahora comete errores con altos valores de certeza. Esta situación es entendible y esperada, debido a que el conjunto de prueba contiene nuevas clases de ataques.

### Segundo nivel de clasificación (K-Medias)

A continuación se añadirá el segundo nivel de clasificación que corresponde al uso de K-Medias para tomar todos aquellos registros clasificados como normal para tratar de corregir los falsos negativos producidos por la red neuronal. El algoritmo de K-Medias será implementado con dos clusters debido a que en la sección de K-Medias se ilustra que con dos clusters la varianza acumulada es la adecuada, adicionalmente se probó que con dos clusters se obtuvieron mejores resultados que con cinco.

```
kmeans.set = testing.set[predictions == "normal", ]  
kmeans.set[,ncol(kmeans.set)] = as.character(kmeans.set[,ncol(kmeans.set)])  
kmeans.set[kmeans.set[,ncol(kmeans.set)] != "normal",ncol(kmeans.set)] = "Attack"  
SumLabels(kmeans.set, ncol(kmeans.set))
```

```
## [1] 4420 9338
```

Se observa como se extrajeron los 4420 **falsos negativos** en conjunto con el resto del **tráfico normal** y ese será el conjunto de datos para la aplicación de **K-Medias**. A continuación se precalcularán los centroides. Las actividades relacionadas con el entrenamiento y predicciones serán cronometradas de igual forma que con el primer nivel de clasificación del modelo.

```
start.time.kmeans.training = Sys.time()  
matrix.centers = FindCentersKmeans(set = kmeans.set, clusters = 2,  
iterations = 100, iter.max = 100)
```

```
#Promediando los centroides
matrix.centers = matrix.centers/100
total.time.kmeans.training = Sys.time() - start.time.kmeans.training
total.time.kmeans.training
```

## Time difference of 6.340442 secs

Ahora se realizarán las predicciones.

```
start.time.kmeans.predictions = Sys.time()
kmeans.model = kmeans(kmeans.set[,1:(ncol(kmeans.set)-1)], centers = matrix.centers,
                      iter.max = 100)

total.time.kmeans.predictions = Sys.time() - start.time.kmeans.predictions
total.time.kmeans.predictions
```

## Time difference of 0.06994319 secs

Ahora se creará la matriz de confusión producto de la clasificación de **K-Medias**.

```
predictions = OrderKmeans(kmeans.model)
confusion.matrix.kmeans.model = table(Real = kmeans.set[,ncol(kmeans.set)],
                                       Prediction = predictions)
confusion.matrix.kmeans.model

##          Prediction
## Real      Attack normal
##   Attack    2334    2086
##   normal    1661    7677
```

Se observa cómo se detectaron 2334 ataques, sin embargo, ahora hay mayor cantidad de **falsos positivos**. Veamos la **tasa de aciertos** y la **tasa de errores**.

```
accuracy.kmeans.model = mean(predictions == kmeans.set[,ncol(kmeans.set)])
accuracy.kmeans.model*100
```

## [1] 72.76494

```
ErrorRate(accuracy.kmeans.model)*100
```

## [1] 27.23506

Se obtiene un 72.76% de acierto, es un número bastante bueno, similar al de la detección de intrusos en el primer nivel. Ahora veamos la tasa de aciertos por etiqueta.

```
AccuracyPerLabel(confusion.matrix.kmeans.model, kmeans.set)
```

## [1] 52.80543 82.21247

Se detecta alrededor de la mitad de los **ataques** presentes y se separa con 82% de certeza el **tráfico normal**. Ahora veamos las medidas binarias de **sensitividad**, **especificidad** y **precisión**.

```

Sensitivity(confusion.matrix.kmeans.model) * 100

## [1] 52.80543

Especifity(confusion.matrix.kmeans.model) * 100

## [1] 82.21247

Precision(confusion.matrix.kmeans.model) * 100

## [1] 58.42303

```

El modelo tiene un desempeño decente en la clasificación del **tráfico normal** y un desempeño intermedio en la detección de ataques. Ahora veamos las estadísticas totales producto de la mezcla de ambos niveles. Empecemos por ver la matriz de confusión.

```

confusion.matrix.two.labels = TwoLevelsCM(attack.normal.confusion.matrix, confusion.matrix.kmeans.model)
confusion.matrix.two.labels

##      [,1] [,2]
## [1,] 10747 2086
## [2,]  2034 7677

```

El resultado total refleja un incremento positivo en la detección de **ataques** y en la reducción de **falsos negativos**. Por otra parte, el incremento de los **falsos positivos** y el decrecimiento de la certeza de clasificación del tráfico normal son aspectos negativos. Veamos la tasa de **aciertos** y de **errores**.

```

accuracy.total = Accuracy(confusion.matrix.two.labels)
accuracy.total * 100

```

```

## [1] 81.72463

ErrorRate(accuracy.total) * 100

## [1] 18.27537

```

La tasa de aciertos mejoró con respecto al primer nivel de clasificación del modelo en un 5%. Una mejora significativa en la detección de intrusos. Ahora veamos cómo quedaron el resto de las medidas de rendimiento concernientes a la **sensitividad**, **especificidad** y **precisión**.

```

Sensitivity(confusion.matrix.two.labels) * 100

## [1] 83.74503

Especifity(confusion.matrix.two.labels) * 100

## [1] 79.05468

```

```
Precision(confusion.matrix.two.labels) * 100
```

```
## [1] 84.08575
```

Se nota un incremento con respecto a la **sensitividad** del primer nivel del 23%. Por otra parte, hubo un decremento con un promedio de alrededor 15% en la **especificidad** y en la **precisión**. En general el modelo híbrido tiene un desempeño bastante bueno, se logra incrementar la cantidad de **ataques** detectados y se obtiene una combinación balanceada entre los **falsos negativos** y **falsos positivos**. Por último el tiempo total para el entrenamiento y las predicciones se imprime a continuación respectivamente.

```
training.time + total.time.kmeans.training
```

```
## Time difference of 273.0243 secs
```

```
total.time.predictions + total.time.kmeans.predictions
```

```
## Time difference of 0.2872779 secs
```

## Conclusiones

El rendimiento del primer nivel con **red neuronal** comparado al rendimiento obtenido en la sección de *análisis sobre el conjunto de entrenamiento* es bastante inferior; sin embargo, es comprensible debido a que en el conjunto de prueba se agregan nuevos tipos de ataques que no estuvieron presentes en el conjunto de entrenamiento. Más allá de eso, el rendimiento es bueno, con 76% de **tasa de aciertos** y buenas medidas de rendimiento para la **sensitividad**, **especificidad** y **precisión**. Por otra parte la **curva ROC** indica que el modelo no es tan certero con respecto a la toma de decisiones, es decir, comete errores con grandes valores de certeza, situación que en la sección de *análisis sobre el conjunto de entrenamiento* no se presentó. Adicionalmente el modelo no comete gran cantidad de **falsos positivos**.

El segundo nivel de **K-Medias** en esta oportunidad tuvo mayor cantidad de ataques debido a que el primer nivel obtuvo un gran número de **falsos negativos**. La **tasa de aciertos** del modelo de **K-Medias** fue del 72.76%, un número similar a la tasa de aciertos del modelo de **red neuronal**. **K-Medias** logró detectar el 50% de los ataques presentes y redujo la cantidad de **falsos negativos** presentes en la entrada; sin embargo, incrementó la cantidad de **falsos positivos** notablemente.

En conjunto, con la inclusión de **K-Medias** se logró un incremento de alrededor del 5% en la **tasa de aciertos** llegando así al 81%, y un incremento del 23% en la **sensitividad**. Por otra parte hubo un decremento de alrededor del 15% en la **especificidad** y **precisión**. Las comparaciones son realizadas con respecto al desempeño del primer nivel del modelo, que corresponde al clasificador de **red neuronal**.

En general el desempeño es bastante bueno, la gran mayoría del tráfico fue clasificado de manera satisfactoria y se produjeron alrededor de 2000 **falsos positivos** y 2000 **falsos negativos** del total de los 22 mil registros presentes en el conjunto de datos. Para un especialista el hecho de que haya mayor cantidad de **falsos positivos** representará más trabajo desde el punto de vista que tendrá que revisar registros que no son una amenaza. Por otra parte, la presencia de **falsos negativos** representa un punto más sensible debido a que los ataques están presentes y no fueron detectados.

## Máquina de soporte vectorial

En esta sección se describirán las actividades realizadas para el entrenamiento y evaluación de la **máquina de soporte vectorial** en el ámbito de la detección de intrusos en redes de computadoras. Esta sección se subdivide en dos grandes partes concernientes al **entrenamiento del modelo** y **evaluación del modelo**. Esto debido a que los pasos y observaciones se harán de manera individual en cada fase.

## Entrenamiento del modelo

Se aplicará el mismo criterio que se propuso en la sección *análisis sobre el conjunto de entrenamiento*; es decir, se usará máquina de soporte vectorial con el **kernel radial**.

Se empezará por establecer el ambiente de trabajo eliminando variables parciales, cargando el archivo de funciones y la vista minable del conjunto de entrenamiento.

```
rm(list = ls())
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv", sep = ",", header = TRUE)
source("../source/functions/functions.R")
```

El paquete utilizado para el entrenamiento de las máquinas de soporte vectorial es **e1071**, a continuación será cargado.

```
library("e1071")
```

Una vez que tenemos nuestro ambiente de trabajo preparado se eliminarán aquella etiquetas del conjunto de datos que no van a ser utilizadas a lo largo del proceso de entrenamiento del modelo. El primer nivel de detección del modelo híbrido posee cinco calses obtetivo que son **DoS**, **normal**, **Probing**, **R2L** y **U2R**; esto con la finalidad de que la salida para el especialista sea más entendible y pueda identificar la(s) falla(s) de seguridad acotándolas dentro de estas cuatro clases de ataques. Dicho esto eliminaremos el resto de las etiquetas.

```
dataset.training$Label_Normal_TypeAttack = NULL
dataset.training$Label_Num_Classifiers = NULL
dataset.training$Label_Normal_or_Attack = NULL
```

Es obligatorio que para el uso de las máquinas de soporte vectorial todas las variables predictoras sean de tipo numérico. Por lo tanto, se transformarán cada una de estas a tipo numérico y la columna objetivo se transformará en tipo factor debido a que se realizarán labores de clasificación.

```
for (i in 1 : (ncol(dataset.training) -1) )
  dataset.training[,i] = as.numeric(dataset.training[,i])

dataset.training[,ncol(dataset.training)] = as.factor(dataset.training[,ncol(dataset.training)])
```

Para acelerar el tiempo de entrenamiento y tener un modelo más preciso es buena práctica escalar el conjunto de datos a rangos similares. En este caso, todas las columnas predictoras tendrán *media* cero (0) y *desviación estandar* uno (1).

```
dataset.training = ScaleSet(dataset.training)
```

Ya se tienen el conjunto de datos listo y el ambiente de trabajo preparado, a continuación se iniciará el proceso de entrenamiento. De igual manera que se realizó en la sección *análisis sobre el conjunto de entrenamiento* el modelo creado será guardado en un objeto debido a que el proceso de entrenamiento es largo y es tedioso tener que esperar a su entrenamiento cada vez que se quiera analizar el modelo. Adicionalmente, en esta oportunidad se calculará el tiempo que tarda el modelo entrenándose. Esto, para poder comparar el tiempo contra la **red neuronal** y luego contra el tiempo de entrenamiento luego de hacer la selección de características y selección de parámetros.

```

start.time = Sys.time()

set.seed(22)
model = svm(Label~.,
            data = dataset.training,
            kernel = "radial",
            scale = FALSE,
            probability = TRUE)

total.time = Sys.time() - start.time

```

Por último, el tiempo y el modelo creado se guardan en una lista y se exportan como un objeto para su posterior uso.

```

list.results = list(total.time, model)
saveRDS(list.results, file = "../source/normal_model/SVM/Real_Model/list_results.rds")

```

### Evaluación del modelo

En esta sección se hará la evaluación de los resultados obtenidos en la sección anterior, adicionalmente se tomará el mejor modelo y las mejores predicciones obtenidas para agregarle el segundo nivel de clasificación correspondiente al algoritmo K-Medias. Se empezará por establecer el ambiente de trabajo eliminando variables parciales, cargando el paquete **e1071**, cargando el archivo de funciones, la lista con información exportada previamente y el conjunto de datos de prueba.

```

rm(list = ls())
library("e1071")
source("../source/functions/functions.R")
results = readRDS("../source/normal_model/SVM/Real_Model/list_results.rds")
testing.set = read.csv("../dataset/NSLKDD_Testing_New.csv", sep = ",", header = TRUE)

```

Se empezará por eliminar las etiquetas innecesarias, transformar las variables predictoras a tipo **numérico** y la columna objetivo a tipo **factor**, y escalar las variables predictoras dentro de la misma *media* y *desviación estándar*.

```

#Eliminando etiquetas
testing.set$Label_Normal_TypeAttack = NULL
testing.set$Label_Num_Classifiers = NULL
testing.set$Label_Normal_or_Attack = NULL

#Cambiando el tipo de dato
for (i in 1 : (ncol(testing.set) -1) )
  testing.set[,i] = as.numeric(testing.set[,i])

testing.set[,ncol(testing.set)] = as.factor(testing.set[,ncol(testing.set)])

#Escalando las variables predictoras
testing.set = ScaleSet(testing.set)

```

Hasta este punto ya se tienen listos el ambiente de trabajo, conjunto de datos y la lista de resultados de la sección anterior. A continuación se extraerá el modelo y el tiempo de entrenamiento del modelo y se visualizará el tiempo correspondiente al entrenamiento del modelo.

```

training.time = results[[1]]
model = results[[2]]
training.time

## Time difference of 20.26872 mins

```

A partir de este punto se empezará con el análisis del modelo. Todos los pasos involucrados con el tiempo de entrenamiento y predicción serán cronometrados y al final será sumados para tener una perspectiva del tiempo necesario para cada fase. Se iniciará con el cálculo de las predicciones.

```

start.time.predictions = Sys.time()

predictions = predict(model, testing.set[, 1:(ncol(testing.set)-1)], type = "class")

total.time.predictions = Sys.time() - start.time.predictions
total.time.predictions

```

```
## Time difference of 30.40421 secs
```

A continuación se creará una matriz de confusión que nos ayude a ver gráficamente el desempeño del modelo durante el proceso de clasificación.

```

confusion.matrix = table(Real = testing.set[,ncol(testing.set)],
                         Prediction = predictions)
confusion.matrix

```

		Prediction				
##	Real	DoS	normal	Probing	R2L	U2R
##	DoS	6125	1266	67	0	0
##	normal	24	9521	158	8	0
##	Probing	173	707	1541	0	0
##	R2L	0	2530	9	215	0
##	U2R	1	177	19	3	0

Si se compara con la matriz de confusión del modelo en la sección *análisis sobre el conjunto de entrenamiento*, se observa una matriz de confusión mucho más desordenada. Sin embargo, a simple vista se observa que la diagonal acumula la mayoría de los registros, adicionalmente se observa que existen más **falsos negativos** que **falsos positivos**, es decir, hubo más errores en los que se clasificó **tráfico normal** como **ataques** que **ataques** que se clasificaron como **tráfico normal**. A continuación veamos la **tasa de aciertos** y la **tasa de errores**.

```

accuracy = mean(testing.set[,ncol(testing.set)] == predictions)
accuracy * 100

```

```
## [1] 77.19127
```

```
ErrorRate(accuracy) * 100
```

```
## [1] 22.80873
```

Ya no se tiene un desempeño tan alto como se tuvo en el *análisis sobre el conjunto de entrenamiento*, y es entendible debido a que en el conjunto de prueba hay clases de ataques que no estuvieron presentes en el conjunto de entrenamiento. Sin embargo, una tasa de aciertos de 77.19% es bastante alta para este escenario y se espera que con la inclusión de **K-Medias** se incremente aún más la tasa de aciertos. Ahora veamos la precisión por etiquetas, recordemos que la salida corresponde a un vector con el siguiente orden: **DoS**, **normal**, **Probing**, **R2L** y **U2R**. En comparación con el modelo de **red neuronal** se tiene un porcentaje de acierto ligeramente mayor, debido a que el modelo de **red neuronal** tuvo una tasa de aciertos de 76.81%.

```
AccuracyPerLabel(confusion.matrix, testing.set)
```

```
## [1] 82.126575 98.043456 63.651384 7.806826 0.000000
```

Para las etiquetas de **DoS**, **normal** y **Probing** el rendimiento es bastante bueno, en especial para **DoS** y **normal**. Sin embargo, para **R2L** y **U2R** es bastante pobre. Esto puede deberse a la poca cantidad de registros usados para el entrenamiento en ambos casos, en particular para la clase **U2R**. Con respecto a la **red neuronal**, este modelo es mejor en la clasificación de las etiquetas **DoS** y **normal**, sin embargo, en las demás el modelo de **red neuronal** tiene un mejor desempeño.

A continuación crearemos una matriz de confusión binaria para poder calcular las medidas de rendimiento binarias correspondientes a **sensitividad**, **especificidad**, **precisión** y la graficación de la **curva ROC**.

```
attack.normal.confusion.matrix = AttackNormalConfusionMatrix(testing.set, predictions)
attack.normal.confusion.matrix
```

```
##          Prediction
## Real      Attack normal
##   Attack     8153    4680
##   normal     190    9521
```

Se nota una baja cantidad de **falsos positivos**, incluso menos cantidad que en el modelo de **red neuronal**, y una alta cantidad de **falsos negativos**, cantidad mayor que en el modelo de **red neuronal**, y una alta tasa de aciertos con respecto a la clasificación de los registros ubicados en la diagonal. Ahora que hay mayor cantidad de **falsos negativos**, el algoritmo de **K-Medias** puede aportar más al tema de la clasificación. Ahora veamos las medidas de rendimiento binarias mencionadas con anterioridad.

```
Sensitivity(attack.normal.confusion.matrix) * 100
```

```
## [1] 63.53152
```

```
Especifity(attack.normal.confusion.matrix) * 100
```

```
## [1] 98.04346
```

```
Precision(attack.normal.confusion.matrix) * 100
```

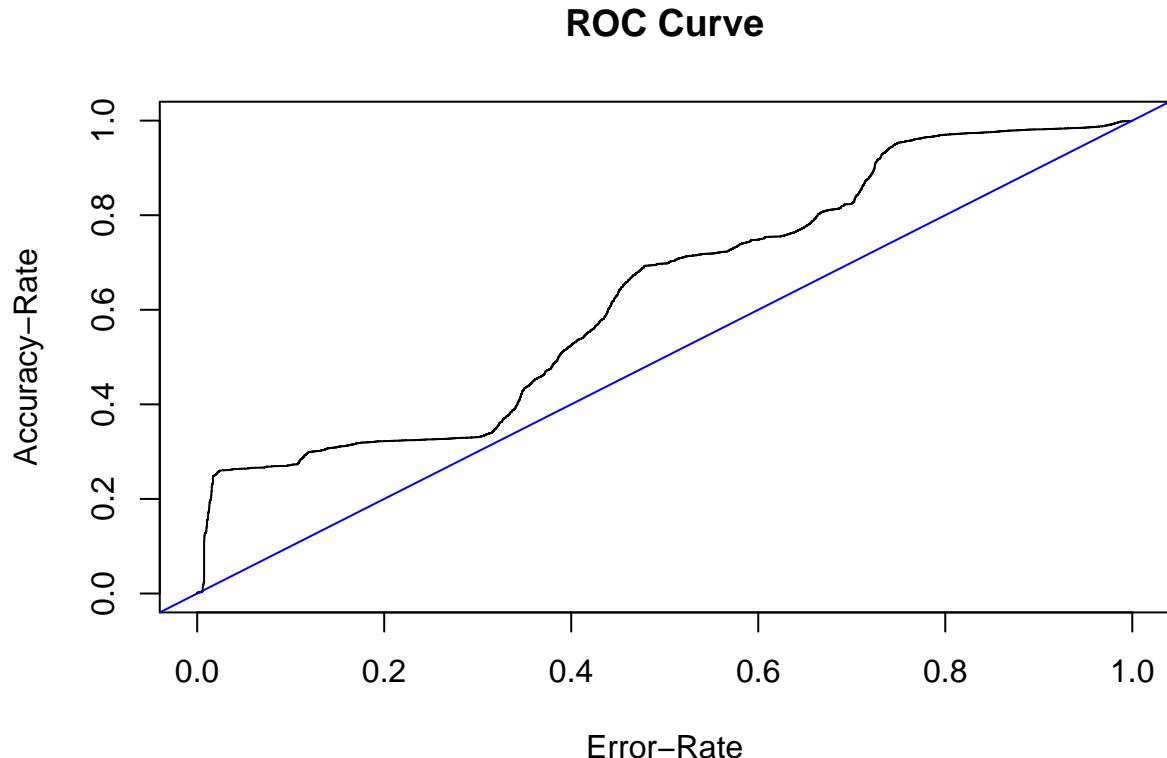
```
## [1] 97.72264
```

La **sensitividad** nos dice que el 63.53% de los ataques fueron detectados de forma correcta; así mismo, la especificidad nos dice que el 98.04% del **tráfico normal** fue clasificado de forma satisfactoria. Por último, la **precisión** nos dice que el 97.72% de los registros clasificados como ataques de verdad eran ataques. Dicho esto el modelo es bastante efectivo a la hora de clasificar el **tráfico normal** y moderadamente bueno

a la hora de clasificar los **ataques**, sin embargo, las decisiones tomadas con respecto a la detección de los ataques es bastante elevada, situación que hace que no tenga tantos **falsos positivos**. Si se compara con el modelo de **red neuronal**, la **red neuronal** detecta mayor cantidad de **ataques**, mientras que la **máquina de soporte vectorial** clasifica mejor el tráfico normal.

El gran problema del modelo recae en la cantidad de **falsos negativos generados**. Se espera que con la inclusión de **K-Medias** esta situación pueda mejorar. A continuación se graficará la **curva ROC**.

```
probabilities = predict(model, testing.set[, 1:(ncol(testing.set)-1)], probability = TRUE)
roc.data = DataROC(testing.set, attr(probabilities, "probabilities"), predictions)
generate_ROC(roc.data$Prob, roc.data$Label, roc.data$Prediction)
```



En comparación con la sección de *análisis sobre el conjunto de entrenamiento*, se observa un desempeño notablemente inferior, en esta ocasión, el desempeño es bastante errático, teniendo su mejor rendimiento al inicio y luego casi pegándose a la línea del azar. Que el desempeño sea inferior es entendible y esperado dada la naturaleza del conjunto de prueba donde hay nuevos tipos de ataques.

#### Segundo nivel de clasificación (K-Medias)

A continuación se añadirá el segundo nivel de clasificación que corresponde al uso de **K-Medias** para tomar todos aquellos registros clasificados como **tráfico normal** y serán pasados al segundo nivel para corregir los **falsos positivos** producidos por el modelo del primer nivel correspondiente a la **máquina de soporte vectorial**. El algoritmo de **K-Medias** será implementado con dos clusters debido a que en la sección de *K-Medias* se ilustra que con dos clusters se acumula la mejor cantidad de varianza, y adicionalmente se probó que con dos clusters se obtuvieron mejores resultados que con cinco clusters.

```
kmeans.set = testing.set[predictions == "normal", ]
kmeans.set[,ncol(kmeans.set)] = as.character(kmeans.set[,ncol(kmeans.set)])
kmeans.set[kmeans.set[,ncol(kmeans.set)] != "normal",ncol(kmeans.set)] = "Attack"
SumLabels(kmeans.set, ncol(kmeans.set))
```

```
## [1] 4680 9521
```

Se observa como se extrajeron los 4680 **falsos negativos** en conjunto con el resto del **tráfico normal**, y ese será el conjunto de datos para la aplicación de **K-Medias**. A continuación, se precalcularán los centroides. Las actividades relacionadas con el entrenamiento y predicción serán cronometradas de igual forma que se hizo en el primer nivel de clasificación del modelo.

```
start.time.kmeans.training = Sys.time()
matrix.centers = FindCentersKmeans(set = kmeans.set, clusters = 2,
                                    iterations = 100, iter.max = 100)

#Promediando los centroides
matrix.centers = matrix.centers/100
total.time.kmeans.training = Sys.time() - start.time.kmeans.training
total.time.kmeans.training
```

```
## Time difference of 7.548652 secs
```

Ahora se realizarán las predicciones.

```
start.time.kmeans.predictions = Sys.time()
kmeans.model = kmeans(kmeans.set[,1:(ncol(kmeans.set)-1)], centers = matrix.centers,
                      iter.max = 100)
total.time.kmeans.predictions = Sys.time() - start.time.kmeans.predictions
total.time.kmeans.predictions
```

```
## Time difference of 0.05890989 secs
```

Ahora se creará la matriz de confusión producto de la clasificación de **k-Medias**.

```
predictions = OrderKmeans(kmeans.model)
confusion.matrix.kmeans.model = table(Real = kmeans.set[,ncol(kmeans.set)],
                                       Prediction = predictions)
confusion.matrix.kmeans.model

##          Prediction
## Real      Attack normal
##   Attack     542    4138
##   normal      80    9441
```

Se observa como se separaron 542 **ataques** de los 4680 iniciales, la cantidad de **falsos negativos** se redujo y la cantidad de **falsos positivos** aumentó en 80. Es una mejora bastante conservadora que tiene un incremento bastante favorable en la detección de los **ataques** sin desordenar de gran manera la clasificación lograda para el **tráfico normal**. Veamos la **tasa de aciertos** y la **tasa de errores**.

```
accuracy.kmeans.model = mean(predictions == kmeans.set[,ncol(kmeans.set)])
accuracy.kmeans.model*100
```

```
## [1] 70.29787
```

```
ErrorRate(accuracy.kmeans.model)*100
```

```
## [1] 29.70213
```

Se obtuvo una **tasa de aciertos** de 70.30%, es un número bastante bueno, similar al obtenido en el primer nivel. Si se compara con el rendimiento obtenido por el modelo de **red neuronal**, entonces se obtiene 2% menos, pero en la **red neuronal** se cometen mayor cantidad de **falsos positivos**. Ahora veamos la tasa de aciertos por etiqueta.

```
AccuracyPerLabel(confusion.matrix.kmeans.model, kmeans.set)
```

```
## [1] 11.58120 99.15975
```

Se detecta solo el 11.58% de los **ataques** presentes, y se clasifica de buena manera el 99.16% del **tráfico normal**. Si se compara con el el modelo de **red neuronal**, la **red neuronal** es mejor detectando \*ataques pero la **máquina de soporte vectorial** clasifica mejor el **tráfico normal**. Ahora veamos las medidas binarias de **sensitividad**, **especificidad** y **precisión**.

```
Sensitivity(confusion.matrix.kmeans.model) * 100
```

```
## [1] 11.5812
```

```
Especifity(confusion.matrix.kmeans.model) * 100
```

```
## [1] 99.15975
```

```
Precision(confusion.matrix.kmeans.model) * 100
```

```
## [1] 87.13826
```

El modelo tiene un desempeño excelente en la detección del **tráfico normal**, por otra parte, el desempeño a la hora de clasificar los **ataques** es bastante pobre, pero acierta con alta probabilidad los ataques detectados, por lo tanto no genera muchos **falsos positivos**. Ahora veamos las estadísticas totales producto de la mezcla de ambos niveles. Empecemos por ver la matriz de confusión.

```
confusion.matrix.two.labels = TwoLevelsCM(attack.normal.confusion.matrix, confusion.matrix.kmeans.model)
confusion.matrix.two.labels
```

```
##      [,1] [,2]
## [1,] 8695 4138
## [2,]  270 9441
```

El resultado total refleja un incremento positivo en la detección de **ataques** aunque no muy grande. La cantidad de **falsos negativos** sigue siendo bastante alta, mientras que la cantidad de **falsos positivos** es bastante baja. Por último, la fortaleza de este modelo es la correcta identificación del tráfico normal, motivo por el cuál existe una gran cantidad de **falsos negativos**. Ahora veamos la **tasa de aciertos** y la **tasa de errores**.

```
accuracy.total = Accuracy(confusion.matrix.two.labels)
accuracy.total * 100
```

```
## [1] 80.44713
```

```
ErrorRate(accuracy.total) * 100
```

```
## [1] 19.55287
```

Se logró un incremento del 3% en la **tasa de aciertos** con respecto al primer nivel de clasificación, una mejora significativa en el área de la detección de intrusos en redes de computadoras. Ahora veamos cómo quedaron el resto de las medidas de rendimiento concernientes a la **sensitividad, especificidad y precisión**.

```
Sensitivity(confusion.matrix.two.labels) * 100
```

```
## [1] 67.75501
```

```
Especificity(confusion.matrix.two.labels) * 100
```

```
## [1] 97.21965
```

```
Precision(confusion.matrix.two.labels) * 100
```

```
## [1] 96.98829
```

Se nota un incremento con respecto a la **sensitividad** del 4%, es decir, se detectaron 4% más de los ataques presentes con la inclusión de **K-Medias**. Por otra parte hubo un decremento de alrededor del 1% con respecto a la **especificidad y precisión**. En general el modelo híbrido tiene un buen desempeño, los puntos altos son la baja generación de **falsos positivos** y la alta eficacia en la clasificación del **tráfico normal**. Los puntos bajos corresponden a la gran cantidad de **falsos negativos** presentes en las predicciones. Por último, el tiempo total para el entrenamiento y las predicciones se mostrará a continuación respectivamente.

```
training.time + total.time.kmeans.training
```

```
## Time difference of 1223.672 secs
```

```
total.time.predictions + total.time.kmeans.predictions
```

```
## Time difference of 30.46312 secs
```

## Conclusiones

El rendimiento del primer nivel con **máquina de soporte vectorial** comparado con el rendimiento obtenido en la sección *análisis sobre el conjunto de entrenamiento* es bastante inferior; sin embargo, es comprensible debido a que en el conjunto de prueba se agregan nuevos tipos de ataques que no estuvieron presentes en el conjunto de entrenamiento. Más allá de eso el rendimiento es bastante bueno con 77% de **tasa de aciertos** y buenas medidas de rendimiento para la **sensitividad, especificidad y precisión**. Por otra parte la **curva ROC** indica que el modelo es bastante variante con respecto a la certeza con la que toma las decisiones,

llegando en un punto a pegarse bastante a la **línea del azar**, situación bastante deteriorada con respecto a la sección de *análisis sobre el conjunto de entrenamiento*. Como aspecto positivo, el modelo no comete gran cantidad de **falsos positivos**, pero si una gran cantidad de **falsos negativos**.

El segundo nivel del modelo, con **K-Medias** tuvo un mejor desempeño comparado con la sección *análisis sobre el conjunto de entrenamiento*, esto debido a que el primer nivel correspondiente a la **máquina de soporte vectorial** tuvo gran cantidad de falsos negativos. La **tasa de aciertos** del modelo **K-Medias** fue del 70.30%, un número similar al del primer nivel. Sin embargo, **K-Medias** solo logró detectar el 11% de los ataques presentes y redujo escasamente la cantidad de **falsos negativos** presentes. Por otra parte, un aspecto positivo fue la no generación excesiva de **falsos positivos**.

En conjunto, la inclusión de **K-Medias** logró un incremento de alrededor del 3% con respecto a los resultados obtenidos en el primer nivel en la **tasa de aciertos**, llegando así a un 80%. Por otra parte el incremento en la **sensitividad** fue de sólo el 4% que corresponde a la proporción de los ataques detectados por **K-Medias**. La **especificidad** y la **precisión** se vieron invariantes, decrementando ambas alrededor de 1%. Estas comparaciones fueron realizadas con respecto al desempeño obtenido por el primer nivel, que corresponde al clasificador de **máquina de soporte vectorial**.

En general el desempeño es bastante bueno, la gran mayoría del tráfico fue clasificado de manera satisfactoria y se produjeron alrededor de 4400 errores en la clasificación, donde 4138 corresponden a **falsos negativos** de los 22 mil registros presentes en el conjunto de prueba. Para un especialista el hecho de que no haya gran cantidad de **falsos positivos** es positivo debido a que no tendrá que invertir tiempo revisando registros que no son una amenaza. Sin embargo, la gran cantidad de falsos negativos representan una gran amenaza debido a que los ataques no fueron detectados y además elimina la posibilidad de poder retroalimentar el modelo tomando los **falsos positivos** y colocándolos como pertenecientes al tráfico normal, para que de esta manera el modelo pueda aumentar su base de conocimientos.

## Conclusiones generales

El modelo de **red neuronal** es más efectivo a la hora de detectar ataques, adicionalmente se observa mediante la **curva ROC** que las decisiones tomadas tienen mayor certeza y son más precisas. Por otra parte, el modelo de **máquina de soporte vectorial** es mejor clasificando el tráfico normal, e incluso individualmente tiene mayor cantidad de **tasa de aciertos**. También se pudo observar que la **máquina de soporte vectorial** es más efectiva detectando las clases **DoS** y **normal**, mientras que la **red neuronal** es mejor detectando el resto de las clases concernientes a **Probing**, **R2L** y **U2R**.

La inclusión de **K-Medias** en los modelos repercutió de manera diferente en ambos modelos. Para la **red neuronal** logró un incremento notable en la cantidad de ataques detectados, pero incrementó notablemente la cantidad de **falsos positivos generados**. Por el contrario, para la **máquina de soporte vectorial** la inclusión de **K-Medias** fue más conservadora, detectando menor cantidad de **ataques** pero sin generar exceso de **falsos positivos**.

Para poder determinar cuál modelo es mejor que otro hay que irse por el tema de prioridades. Es decir, ¿Es más importante tener más cantidad de ataques detectados con un mayor número de falsos positivos presentes o es mejor un enfoque más conservador con menor cantidad de ataques detectados pero con menor cantidad de **falsos positivos** presentes? Particularmente me parece que la red neuronal es mejor debido a que el objetivo es la detección de ataques. Adicionalmente, en este caso los **falsos positivos** incrementan el trabajo del especialista para examinar los posibles ataques, y en caso de que una no sea correcta, esta puede ser etiquetada y ser usada para la retroalimentación del modelo, es decir, hay una curva de aprendizaje mucho más rápida que en el modelo híbrido de la máquina de soporte vectorial.

Hasta este punto se han usado los parámetros por defecto, queda como tarea pendiente aún realizar la selección de características y la selección de parámetros y analizar el impacto sobre ambos enfoques.

## Selección de características

En esta sección se realizan las actividades concernientes a la selección de características. La idea principal detrás de la reducción de características es la de quitar aquellas variables predictoras que puedan introducir ruido al modelo, adicionalmente al haber menor cantidad de dimensiones el modelo es entrenado de forma más rápida, y las predicciones también son hechas con mayor velocidad. El conjunto de datos **NSL-KDD** quedó con 40 variables predictoras luego de realizar el pre-procesamiento, y en esta sección se reducirá su número y se analizará su impacto para los modelos híbridos basados en **red neuronal** y en **máquina de soporte vectorial**.

Se aplicarán dos métodos para la selección de características. El primer método, que es uno de los más populares y ampliamente usados en el área de **aprendizaje automático** es el **Análisis de Componentes Principales - PCA**. Con la técnica de PCA se crea un nuevo espacio de variables predictoras basándose en combinaciones lineales entre las mismas. Cómo ventaja para este enfoque se tiene una manera efectiva de visualizar y obtener aquellas nuevas variables predictoras que acumulan mayor cantidad de varianza. Por otra parte, se pierde interpretabilidad de los datos, debido a que ya no hay variables predictoras con un nombre que se pueda asociar a un evento producido en el ámbito del problema.

Como segunda técnica se usará la **Reducción Gradual de Características - GFR** que es una técnica propuesta por Li en su trabajo *An Efficient Intrusion Detection System Based on Support Vector Machines and Gradually Feature Removal Method*. Esta técnica tuvo buenos resultados en dicha publicación, adicionalmente es sencilla de implementar y de esta manera se puede visualizar cuales son las variables predictoras más importantes ya que en esta se mantiene la interpretabilidad de los datos. Por otra parte, habrá que compararla con PCA para saber cuál de estas tiene mejor desempeño.

Comenzaremos por la implementacion de PCA y posteriormente con GFR.

### PCA

En esta sección se describirán las actividades concernientes a la implementación y análisis de la aplicación de PCA sobre el conjunto de datos **NSL-KDD** para la reducción de características. Estas actividades corresponden al **análisis exploratorio** y posteriormente se calculará el error producido por cada uno de los modelos basados de **red neuronal** y **máquina de soporte vectorial**.

#### Análisis exploratorio

Acá se aplicará PCA sobre el conjunto de datos y se verá con cuántas variables predictoras se acumula una cantidad suficiente de varianza acumulada. También se verá si este número de variables corresponde a una reducción significativa.

Empezaremos las actividades limpiando el ambiente de trabajo, cargando el conjunto de datos de entrenamiento y el archivo de funciones.

```
rm(list = ls())
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv",
                           sep = ",", header = TRUE)
source("../source/functions/functions.R")
```

Para probar el error de las características, se usarán los clasificadores **red neuronal** y **máquina de soporte vectorial**, debido a esto se usaran 5 clases objetivo y motivado por esto es necesario eliminar aquellas etiquetas innecesarias, transformar las columnas predictoras a tipo **numérico**, la columna objetivo a tipo **factor** y escalar el conjunto de datos para que estos tengan *media* cero (0) y *desviación estándar* uno (1).

```

#Eliminando columnas innecesarias
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL

#Cambiando el tipo de dato de las columnas
for (i in 1:(ncol(dataset)-1))
  dataset[,i] = as.numeric(dataset[,i])

dataset[,ncol(dataset)] = as.factor(dataset[,ncol(dataset)])

#Escalando las variables predictoras
dataset = ScaleSet(dataset)

```

Ya se tiene el ambiente de trabajo listo, y ahora podemos aplicar PCA.

```
pca = prcomp(dataset[,-41], scale. = TRUE)
```

Se utilizó la función **prcomp** perteneciente a la biblioteca **stats**. se observa que como parámetros se pasaron todas las variables predictoras (se dejó por fuera la variable objetivo), y se pidió que se escalara el conjunto de datos. El escalamiento de los datos juega un rol fundamental en PCA debido a que las combinaciones lineales ameritan que los valores estén unificados con respecto a su rango para poder tener éxito. De otra manera, las combinaciones lineales podrían no tener sentido. A continuación veamos un resumen del objeto **pca**.

```
summary(pca)
```

```

## Importance of components:
##                 PC1     PC2     PC3     PC4     PC5     PC6
## Standard deviation   2.7842  2.2758  1.67855  1.45803  1.39380  1.29213
## Proportion of Variance 0.1938  0.1295  0.07044  0.05315  0.04857  0.04174
## Cumulative Proportion 0.1938  0.3233  0.39372  0.44686  0.49543  0.53717
##                 PC7     PC8     PC9     PC10    PC11    PC12
## Standard deviation   1.2555  1.1455  1.05883  1.04509  1.02839  1.00350
## Proportion of Variance 0.0394  0.0328  0.02803  0.02731  0.02644  0.02518
## Cumulative Proportion 0.5766  0.6094  0.63741  0.66471  0.69115  0.71633
##                 PC13    PC14    PC15    PC16    PC17    PC18
## Standard deviation   1.0001  1.0000  0.99726  0.99348  0.96406  0.94985
## Proportion of Variance 0.0250  0.0250  0.02486  0.02468  0.02324  0.02256
## Cumulative Proportion 0.7413  0.7663  0.79119  0.81587  0.83910  0.86166
##                 PC19    PC20    PC21    PC22    PC23    PC24
## Standard deviation   0.87354 0.83652 0.7848  0.77298 0.69884 0.66839
## Proportion of Variance 0.01908 0.01749 0.0154  0.01494 0.01221 0.01117
## Cumulative Proportion 0.88073 0.89823 0.9136  0.92856 0.94077 0.95194
##                 PC25    PC26    PC27    PC28    PC29    PC30
## Standard deviation   0.64268 0.59476 0.56037 0.4857  0.37344 0.36283
## Proportion of Variance 0.01033 0.00884 0.00785 0.0059  0.00349 0.00329
## Cumulative Proportion 0.96227 0.97111 0.97896 0.9849  0.98834 0.99164
##                 PC31    PC32    PC33    PC34    PC35    PC36
## Standard deviation   0.31398 0.25630 0.22109 0.20809 0.16992 0.14567
## Proportion of Variance 0.00246 0.00164 0.00122 0.00108 0.00072 0.00053

```

```

## Cumulative Proportion  0.99410 0.99574 0.99696 0.99805 0.99877 0.99930
##                               PC37     PC38     PC39     PC40
## Standard deviation      0.11987 0.09491 0.06388 0.02346
## Proportion of Variance 0.00036 0.00023 0.00010 0.00001
## Cumulative Proportion   0.99966 0.99988 0.99999 1.00000

```

Se observa que las componentes fueron enumeradas en las columnas de la forma **PCX** donde la *X* corresponde a un número en el rango [1,40] debido a que teníamos 40 variables predictoras inicialmente. Adicionalmente las filas corresponden a tres medidas que son: *desviación estándar* que mide la desviación estándar que se logra sin dicha componente. La *proporción de varianza* dice cuál es la varianza lograda por dicha componente individualmente. Por último, la *proporción acumulada* tiene la sumatoria de todas las proporciones de varianza hasta cierto punto; es decir, la *proporción acumulada* hasta la **componente principal 3** es la sumatoria de la proporción de varianza desde PC1 hasta PC3.

Las primeras componentes al ser las que mayor cantidad de varianza acumulan son las más relevantes. A continuación colocaremos en un **dataframe** las siguientes medidas: *desviación estándar*, *varianza por componente*, *porcentaje de varianza acumulada* y *varianza acumulada*.

```

std.deviation = pca$sdev
PC.variance = std.deviation^2
PR.variance = PC.variance/sum(PC.variance)
cum.variance = cumsum(PR.variance) * 100
summary.pca = data.frame(std_deviation = std.deviation,
                           PC_variance = PC.variance,
                           PR_variance = PR.variance,
                           cum_variance = cum.variance)
summary.pca

```

	std_deviation	PC_variance	PR_variance	cum_variance
## 1	2.78419677	7.7517516596	0.1937937915	19.37938
## 2	2.27583578	5.1794284776	0.1294857119	32.32795
## 3	1.67854984	2.8175295769	0.0704382394	39.37177
## 4	1.45803491	2.1258658076	0.0531466452	44.68644
## 5	1.39380286	1.9426864002	0.0485671600	49.54315
## 6	1.29213141	1.6696035730	0.0417400893	53.71716
## 7	1.25545128	1.5761579268	0.0394039482	57.65756
## 8	1.14545241	1.3120612185	0.0328015305	60.93771
## 9	1.05883068	1.1211224006	0.0280280600	63.74052
## 10	1.04508883	1.0922106575	0.0273052664	66.47104
## 11	1.02838966	1.0575852834	0.0264396321	69.11501
## 12	1.00350121	1.0070146699	0.0251753667	71.63254
## 13	1.00006515	1.0001303068	0.0250032577	74.13287
## 14	0.99998680	0.9999735905	0.0249993398	76.63280
## 15	0.99725592	0.9945193708	0.0248629843	79.11910
## 16	0.99348388	0.9870102190	0.0246752555	81.58663
## 17	0.96406249	0.9294164802	0.0232354120	83.91017
## 18	0.94985274	0.9022202273	0.0225555057	86.16572
## 19	0.87354132	0.7630744328	0.0190768608	88.07341
## 20	0.83651728	0.6997611537	0.0174940288	89.82281
## 21	0.78476561	0.6158570680	0.0153964267	91.36245
## 22	0.77297793	0.5974948872	0.0149373722	92.85619
## 23	0.69884224	0.4883804825	0.0122095121	94.07714
## 24	0.66838599	0.4467398298	0.0111684957	95.19399
## 25	0.64267812	0.4130351632	0.0103258791	96.22658

```

## 26 0.59475922 0.3537385252 0.0088434631 97.11092
## 27 0.56037118 0.3140158602 0.0078503965 97.89596
## 28 0.48574705 0.2359501968 0.0058987549 98.48584
## 29 0.37344455 0.1394608348 0.0034865209 98.83449
## 30 0.36282705 0.1316434713 0.0032910868 99.16360
## 31 0.31398185 0.0985846017 0.0024646150 99.41006
## 32 0.25630325 0.0656913559 0.0016422839 99.57429
## 33 0.22109079 0.0488811385 0.0012220285 99.69649
## 34 0.20809316 0.0433027639 0.0010825691 99.80475
## 35 0.16991893 0.0288724421 0.0007218111 99.87693
## 36 0.14567075 0.0212199676 0.0005304992 99.92998
## 37 0.11987284 0.0143694970 0.0003592374 99.96590
## 38 0.09490743 0.0090074209 0.0002251855 99.98842
## 39 0.06387895 0.0040805209 0.0001020130 99.99862
## 40 0.02346359 0.0005505401 0.0000137635 100.00000

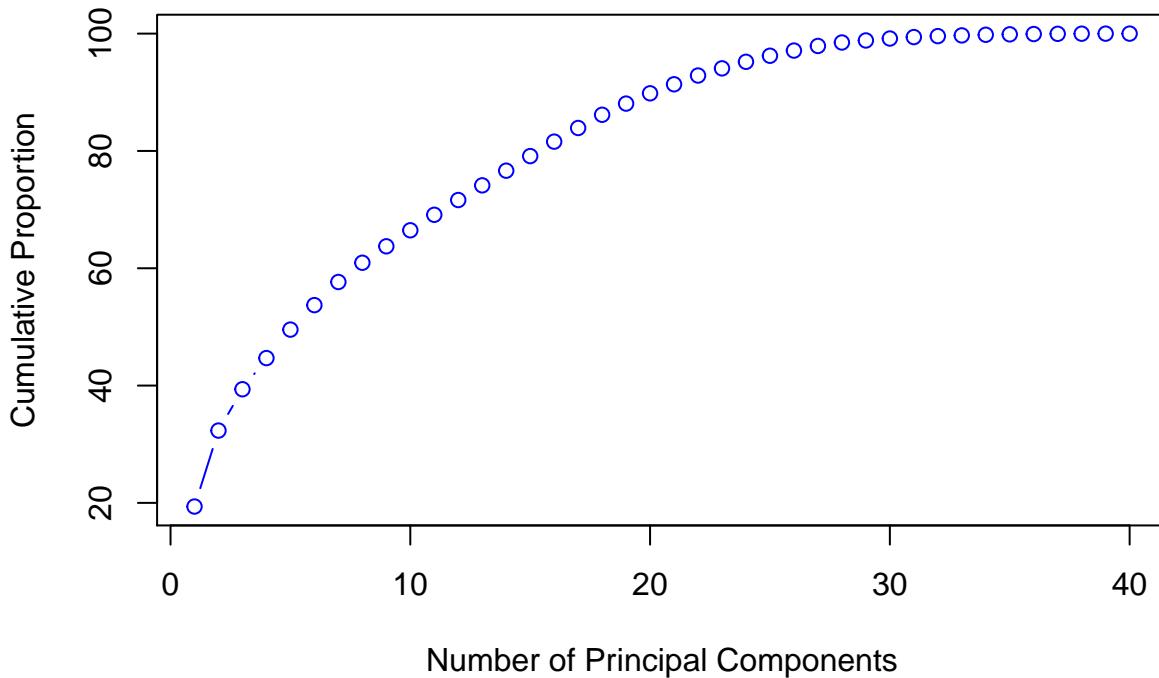
```

De esta manera podemos ver por lo menos en la primera fila que la componente número 1 tiene una *desviación estándar* de 2.78, una varianza de 7.75, un *porcentaje de varianza acumulada* de 19.38% y una varianza acumulada de 19.38%. Anteriormente se mencionó que la selección de las componentes principales debería tener una varianza acumulada de alrededor 95%, este número se alcanza con 24 componentes. Lo que nos dice que teóricamente con 24 variables predictoras de nuestras componentes principales se puede tener una buena selección de características. Ahora grafiquemos la varianza acumulada en función del número de componentes, de esta manera, se puede tener una vista gráfica de a partir de cuál cantidad de componentes principales la varianza se estabiliza.

```

plot(summary.pca$cum_variance,
      ylab = "Cumulative Proportion",
      xlab = "Number of Principal Components",
      type = "b", col = "blue")

```



Se observa que a partir de aproximadamente 24 componentes, la varianza acumulada crece de manera bastante

lenta y aparentemente se estabiliza en ese punto. Dicho esto, 24 debería ser un buen número de variables predictoras a usar, reduciendo así en 16 la dimensionalidad del conjunto de datos.

Se unificará el nuevo espacio de variables predictoras con las etiquetas correspondientes a cada registro.

```
dataset.pca = as.data.frame(pca$x)
dataset.pca = data.frame(dataset.pca,
                        Label = dataset$Label)
```

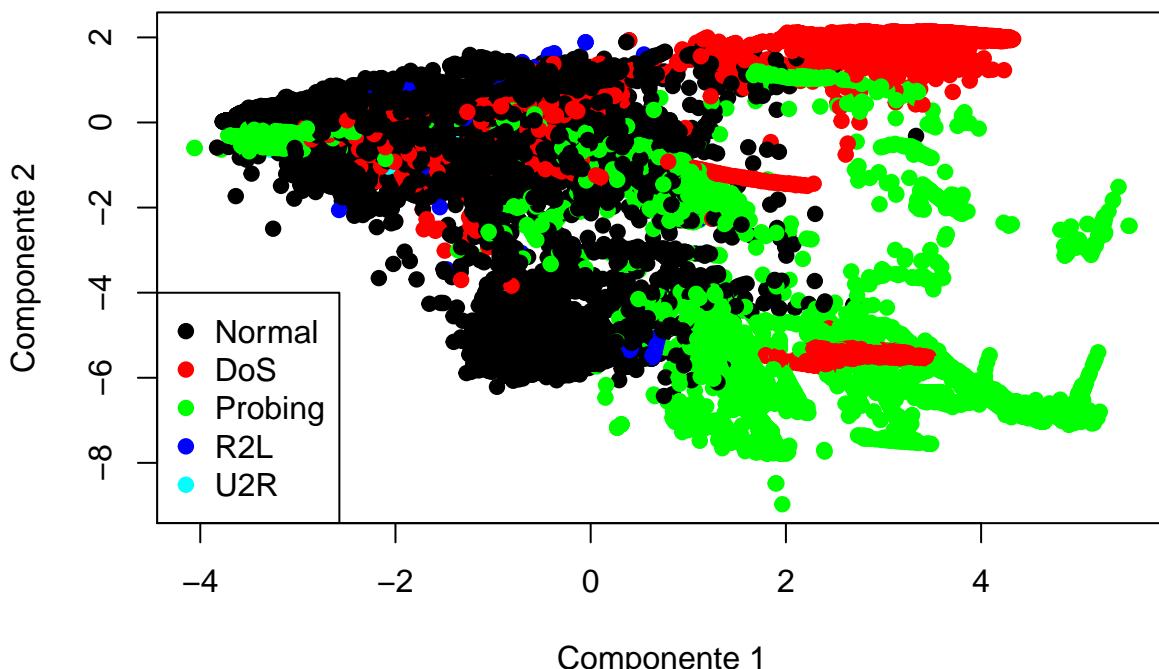
Para terminar con el análisis exploratorio se graficarán las primeras dos componentes principales para visualizar si existe una separación notable entre los registros clasificados como **ataques** y los clasificados como **tráfico normal**.

```
colors = as.character(dataset.pca[, ncol(dataset.pca)])
colors[colors == "normal"] = "black"
colors[colors == "DoS"] = "red"
colors[colors == "Probing"] = "green"
colors[colors == "R2L"] = "blue"
colors[colors == "U2R"] = "cyan"

plot(x = dataset.pca[,1], y = dataset.pca[,2], col = colors,
      main = "Gráfico de las Dos Componentes Principales",
      xlab = "Componente 1", ylab = "Componente 2", pch = 19)

legend("bottomleft", legend = c("Normal", "DoS", "Probing", "R2L", "U2R"),
       col = c("black", "red", "green", "blue", "cyan"), pch = 19)
```

## Gráfico de las Dos Componentes Principales



En la gráfica se observa como los ataques **DoS** y **Probing** en su mayoría poseen una separación bastante marcada con respecto al **tráfico normal**, adicionalmente se observa que estas fronteras poseen una forma no

lineal. Los ataques **R2L** y **U2R** están escondidos dentro del conglomerado de puntos del **tráfico normal**, situación que posiblemente conlleve a que esta clase de \*ataques sea más difícil de detectar que las clases DoS y Probing\*\*.

Adicionalmente, cómo se usará **validación cruzada de 10 conjuntos**, se dividirá el conjunto de datos en 10 subconjuntos de manera estratificada.

```
cv.sets = CVSet(dataset.pca, k = 10, seed = 22)
```

## Efecto de PCA sobre SVM

En esta sección se evaluará el efecto de la aplicación de PCA sobre SVM. Para esto se entrenarán modelos haciendo uso desde [1,40] variables predictoras y se calculará la tasa de aciertos en cada iteración. Para validar el modelo se hará uso de la técnica de validación de modelos de **validación cruzada de 10 conjuntos**. Esta sección se dividirá en dos partes: **entrenamiento y análisis**.

### Entrenamiento

A continuación se describen todas las tareas realizadas en el proceso de entrenamiento de los modelos en el rango de [1,40] variables predictoras. Cabe destacar que se hará uso de las variables parciales utilizadas en la *sección de PCA*, ya que son necesarias para la elaboración de los diferentes modelos.

Empezaremos por crear una matriz para almacenar los resultados de cada iteración. La matriz será de 40x10, donde las 40 filas corresponden al número de componentes y las 10 columnas a cada iteración corresponden a una iteración en el proceso de validación cruzada. Con esta matriz luego se pueden calcular medidas como la *media* por componente y la *desviación estándar o varianza* dentro de cada componente.

```
results = matrix(nrow = 40, ncol = 10)
```

El siguiente segmento de código es el encargado de ejecutar las 400 iteraciones correspondientes al entrenamiento de los modelos de SVM utilizando **validación cruzada de 10 conjuntos**.

```
for (i in 1:40)
{
  results.cv = vector(mode = "numeric", length = 10)

  for (j in 1:10)
  {
    data.cv.testing = cv.sets[[j]]
    data.cv.training = cv.sets
    data.cv.training[[j]] = NULL
    data.cv.testing = as.data.frame(data.cv.testing)
    data.cv.training = do.call(rbind, data.cv.training)

    data.training.pca = as.data.frame(data.cv.training[,1:i])
    colnames(data.training.pca) = names(data.cv.training)[1:i]
    data.training.pca = data.frame(data.training.pca,
                                    Label = data.cv.training$Label)

    data.testing.pca = as.data.frame(data.cv.testing[,1:i])
    colnames(data.testing.pca) = names(data.cv.testing)[1:i]
    data.testing.pca = data.frame(data.testing.pca,
                                 Label = data.cv.testing$Label)
```

```

model = svm(Label ~ .,
            data = data.training.pca,
            kernel = "radial",
            scale = FALSE)

if(i==1)
  prediction = predict(model, data.frame(PC1 = data.testing.pca[,1]), type = "class")
else
  prediction = predict(model, data.testing.pca[,1:i], type = "class")

results.cv[j] = mean(prediction == data.testing.pca[,ncol(data.testing.pca)])
}

results[i,] = results.cv
cat(i, " ")
}

```

Una vez que se acaba el proceso, la matriz es exportada como un objeto para su posterior análisis. Esto es debido a que el proceso para el entrenamiento de los 400 modelos llevó alrededor de 16 horas, y es tedioso tener que esperar todo ese tiempo cada vez que se quiera analizar los resultados.

```
saveRDS(results, file = "../source/feature_selection/SVM/results_PCA.rds")
```

## Análisis

En esta sección se realizará el análisis de los resultados obtenidos en la fase de entrenamiento. Se cargarán los resultados, se calcularán la *desviación estándar* y la *media* de los resultados por cada componente y se graficarán para poder decidir un buen número de componentes a elegir para nuestro modelo definitivo.

Se empezarán las tareas preparando el ambiente de trabajo, esto incluye la eliminación de variables parciales y la carga del objeto de resultados de la sección anterior.

```
rm(list = ls())
results = readRDS("../source/feature_selection/SVM/results_PCA.rds")
```

En la variable **results** se tiene una matriz con los resultados de la **eficacia** producto de la validación cruzada sobre la combinación de componentes principales en el intervalo [1,40]. A continuación se crearán dos vectores en los cuales de almacenarán los resultados producto del cálculo de la *desviación estándar* y la *media* de la **eficacia** por cada una de las componentes.

```
sd.results = apply(results, 1, sd)
mean.results = apply(results, 1, mean)
```

Para seleccionar el número de componentes se usará un criterio similar al del **codo de jambu**, es decir, se buscará el punto donde la **eficacia** empieza a suavizarse conforme el número de componentes principales son agregadas, adicionalmente, se verificará que la *desviación estándar* sea poca para dicho número de componentes.

```
#Dividiendo la pantalla en dos columnas
par(mfrow = c(1,2))

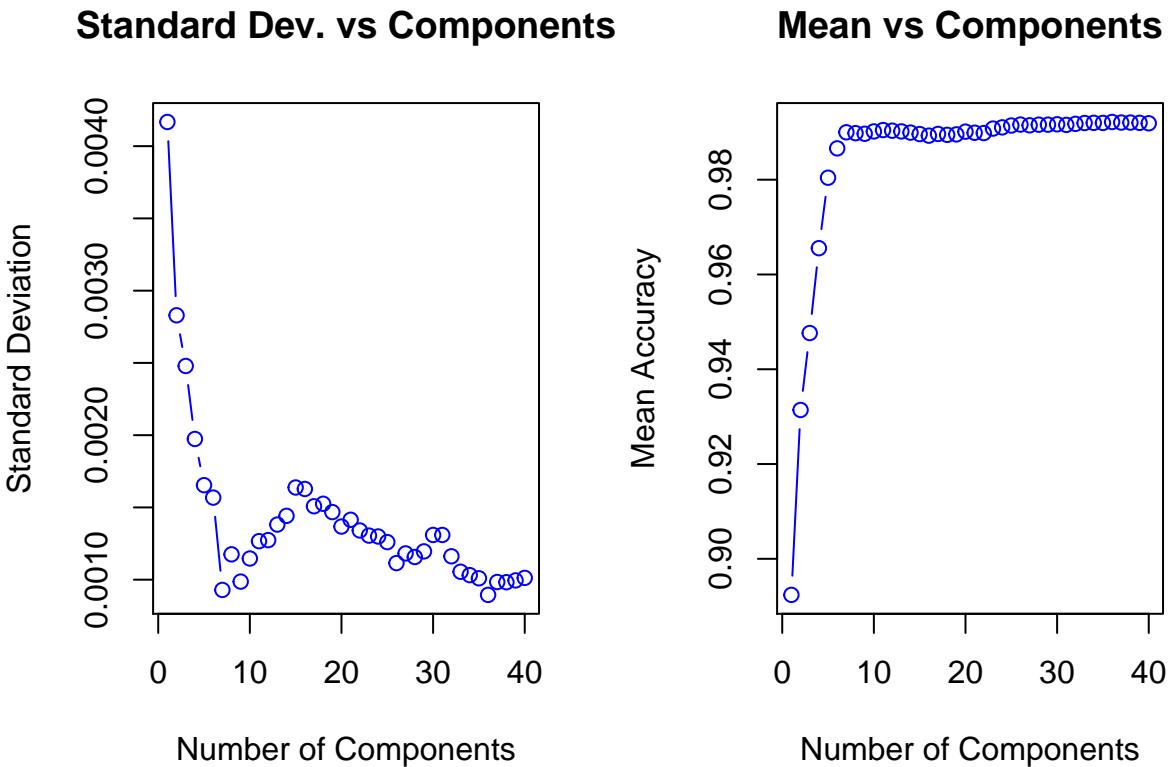
#Graficando Desviación Estándar vs Número de Componentes
```

```

plot(sd.results, col = "blue", type = "b",
      main = "Standard Dev. vs Components",
      xlab = "Number of Components", ylab = "Standard Deviation")

#Graficando Media de Eficacia vs Número de Componentes
plot(mean.results, col = "blue", type = "b",
      main = "Mean vs Components",
      xlab = "Number of Components", ylab = "Mean Accuracy")

```



En las gráficas se observa que con 7 componentes principales que logra una **tasa de aciertos** de alrededor del 99%. A partir de ese punto la mejora obtenida es mínima; adicionalmente, se observa que la *desviación estándar* para dicho número de componentes principales es bastante bajo.

Por lo anterior, se puede pensar que con 7 componentes principales se lograría una buena tasa de aciertos en la **detección de intrusos** y se reduciría la dimensionalidad del conjunto de datos en un 82.5%.

### Efecto de PCA sobre NN

En esta sección se evaluará el efecto de la aplicación de PCA sobre NN. Para esto se entrenarán modelos haciendo uso desde [1,40] variables predictoras y se calculará la tasa de aciertos en cada iteración. Para validar el modelo se hará uso de la técnica de validación de **modelos de validación cruzada de 10 conjuntos**. Esta sección se dividirá en dos partes **entrenamiento** y **análisis**.

### Entrenamiento

A continuación se describen todas las tareas realizadas en el proceso de entrenamiento de los modelos en el rango [1,40] variables predictoras. Cabe destacar que se hará uso de las variables parciales utilizadas en la **sección de PCA**, ya que son necesarias para la elaboración de los diferentes modelos.

Empezaremos por crear una matriz para almacenar los resultados de cada iteración. La matriz será de 40x10, donde las 40 filas corresponden al número de componentes y las 10 columnas a cada iteración durante el proceso de **validación cruzada**. Con esta matriz luego se pueden calcular medidas como la *media*, la *desviación estándar* o *varianza* por número de componentes.

```
results = matrix(nrow = 40, ncol = 10)
```

El siguiente segmento de código es el encargado de ejecutar las 400 iteraciones correspondientes al entrenamiento de los modelos de NN utilizando **validación cruzada de 10 conjuntos**.

```
for (i in 1:40)
{
  results.cv = vector(mode = "numeric", length = 10)

  for (j in 1:10)
  {
    data.cv.testing = cv.sets[[j]]
    data.cv.training = cv.sets
    data.cv.training[[j]] = NULL
    data.cv.testing = as.data.frame(data.cv.testing)
    data.cv.training = do.call(rbind, data.cv.training)

    data.training.pca = as.data.frame(data.cv.training[,1:i])
    colnames(data.training.pca) = names(data.cv.training)[1:i]
    data.training.pca = data.frame(data.training.pca,
                                    Label = data.cv.training$Label)

    data.testing.pca = as.data.frame(data.cv.testing[,1:i])
    colnames(data.testing.pca) = names(data.cv.testing)[1:i]
    data.testing.pca = data.frame(data.testing.pca,
                                  Label = data.cv.testing$Label)

    model = nnet(Label ~ .,
                 data = data.training.pca,
                 size = 20,
                 maxit = 100)

    if(i==1)
      prediction = predict(model, data.frame(PC1 = data.testing.pca[,1]), type = "class")
    else
      prediction = predict(model, data.testing.pca[,1:i], type = "class")

    results.cv[j] = mean(prediction == data.testing.pca[,ncol(data.testing.pca)])
  }

  results[i,] = results.cv
  cat(i, " ")
}
```

una vez que se acaba el proceso, la matriz es exportada como un objeto para su posterior análisis. Esto es debido a que el proceso de entrenamiento para los 400 modelos llevó alrededor de 13 horas, y es tedioso tener que esperar todo ese tiempo cada vez que se quieran analizar los resultados.

```
saveRDS(results, file = "../source/feature_selection/NN/results_PCA.rds")
```

## Análisis

En esta sección se realizará el análisis de los resultados obtenidos en la fase de entrenamiento. Se cargarán los resultados, se calculará la *desviación estándar* y la *media* de los resultados por cada número de componentes y se graficarán para poder decidir un buen número de componentes a elegir para nuestro modelo definitivo.

Se empezarán las tareas preparando el ambiente de trabajo, esto incluye la eliminación de variables parciales y la carga del objeto de los resultados de la sección anterior.

```
rm(list = ls())
results = readRDS("../source/feature_selection/NN/results_PCA.rds")
```

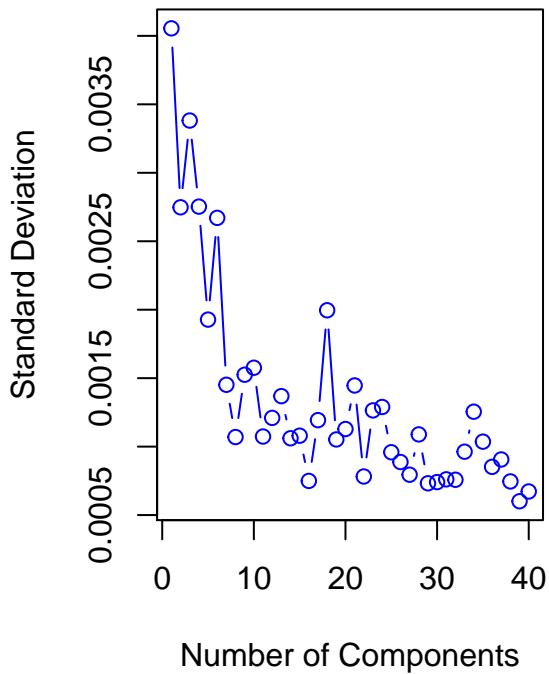
En la variable **results** se tiene una matriz con los resultados de la **eficacia** producto de la aplicación de **validación cruzada** sobre la combinación de componentes principales en el intervalo [1,40]. A continuación se crearán dos vectores en los cuales se almacenarán los resultados producto del cálculo de la *desviación estándar* y la *media* de la **eficacia** por cada una de las componentes.

```
sd.results = apply(results, 1, sd)
mean.results = apply(results, 1, mean)
```

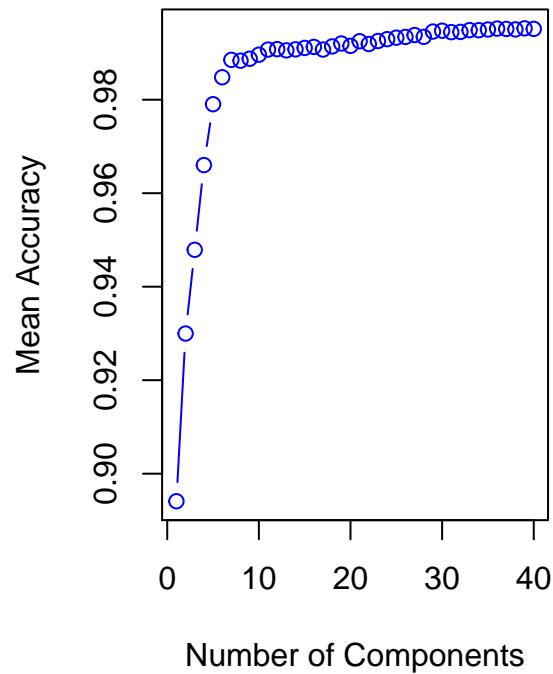
Para seleccionar el número de componentes se usará un criterio similar al del **codo de jambú**. Es decir, se buscará el punto donde la **eficacia** empieza a suavizarse conforme el número de componentes principales son agregadas; adicionalmente, se verificará que la *desviación estándar* sea poca para dicho número de componentes.

```
#Dividiendo la pantalla en dos columnas
par(mfrow = c(1,2))
#Graficando Desviación Estándar vs Número de Componentes
plot(sd.results, col = "blue", type = "b",
main = "Standard Dev. vs Components",
xlab = "Number of Components", ylab = "Standard Deviation")
#Graficando Media de Eficacia vs Número de Componentes
plot(mean.results, col = "blue", type = "b",
main = "Mean vs Components",
xlab = "Number of Components", ylab = "Mean Accuracy")
```

**Standard Dev. vs Components**



**Mean vs Components**



En las gráficas se observa que con 7 componentes principales se logra una **tasa de aciertos** de alrededor del 99%. A partir de ese punto, la mejora obtenida es mínima; adicionalmente, se observa que la **desviación estándar** para dicho número de componentes principales es bastante bajo.

Por lo anterior se puede pensar que con 7 componentes principales se lograría una buena tasa de aciertos en la **detección de intrusos**, y se reduciría la dimensionalidad del conjunto de datos en un 82.5%.

## Conclusión

Una buena medida para seleccionar el número de componentes principales según **Andrew Ng** experto en el área de **aprendizaje automático** es elegir el número de componentes principales que logren capturar varianza en el rango [95%, 99%]. En el análisis exploratorio se observó que la medida de 95% es alcanzada con el uso de 24 componentes principales. Sin embargo, en las secciones donde se realizó el análisis de PCA sobre SVM y NN, se puede notar que con 7 componentes principales se logra un excelente rendimiento reduciendo en 82.5% la dimensionalidad del conjunto de datos. La fase de análisis fue realizada haciendo uso de la técnica de validación de modelos de **validación cruzada de 10 conjuntos** y quedaría por ver el rendimiento de estos algoritmos utilizando el conjunto de pruebas para medir la eficacia de los mismos.

## GFR

En esta sección se describen las actividades concernientes a la implementación y análisis de *GFR* sobre el conjunto de datos NSL-KDD para la reducción de características. estas actividades corresponden al análisis de los resultados obtenidos para los modelos de *red neuronal* y *máquina de soporte vectorial*.

### Efecto de GFR sobre SVM

En esta sección se evaluará el efecto de la aplicación de GFR sobre SVM. Para esto se entrenarán modelos haciendo uso desde [1,40] variable predictoras y se calculará la tasa de aciertos en cada iteración. Para validar

el modelo se hará uso de la técnica de validación de modelos de *validación cruzada de 10 conjuntos*. Esta sección se divide en dos partes: *entrenamiento y análisis*.

## Entrenamiento

Acá se describen todas las tareas realizadas en el proceso de entrenamiento de los modelos en el rango [1,40] variables predictoras. Se iniciará preparando el ambiente de trabajo: limpiar variables parciales, cargar paquetes necesarios y archivo de funciones.

```
rm(list = ls())
require("e1071")
source("functions/functions.R")
```

Se eliminan las etiquetas a no ser usadas.

```
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL
```

Se transforman las variables predictoras a tipo *numérico*.

```
for (i in 1:(ncol(dataset)-1))
  dataset[,i] = as.numeric(dataset[,i])
```

Se transforma la variable objetivo a tipo *factor* esto es debido a que se realizarán labores de *clasificación*. Es importante recordar que la variable objetivo posee cinco clases: DoS, normal, Probing, R2L y U2R.

```
dataset[,ncol(dataset)] = as.factor(dataset[,ncol(dataset)])
```

Se escala el conjunto de datos para que todas las variables predictoras tengan *media cero* (0) y *desviación estándar* uno (1).

```
dataset = ScaleSet(dataset)
```

En este punto se tiene todo listo para aplicar el algoritmo GFR, este fue comprimido en la función **GFR** que recibe como parámetros un **dataframe** y el tipo de algoritmo que se desea usar **SVM** o **NN**.

```
results = GFR(dataset, "NN")
```

*GFR* retorna una matriz de dimensiones 41x10 donde 41 son la cantidad de resultados por característica eliminada y 10 son los resultados obtenidos por iteración durante el proceso de *validación cruzada de 10 conjuntos*. Por último dicha matriz será almacenada en un objeto para su posterior análisis. El tiempo de entrenamiento de este método fue de 12 días, un tiempo bastante elevado y más si se compara con el método de reducción de características PCA.

```
saveRDS(results, "../source/feature_selection/SVM/results_GFR.rds")
```

## Análisis

En esta sección se describen las actividades realizadas para la fase de análisis. Esta empezará limpiando el ambiente de trabajo de variables parciales y cargando el archivo de funciones.

```
rm(list = ls())
source("../source/functions/functions.R")
```

A continuación se cargará el objeto con los resultados obtenidos en la sección anterior.

```
svm.gfr = readRDS("../source/feature_selection/SVM/results_GFR.rds")
```

Las características más importantes extraídas en el proceso anterior se ilustran a continuación en orden descendente de importancia; es decir, la primera representa la más importante y las siguientes implican menor importancia.

```
rownames(svm.gfr)[-nrow(svm.gfr)]
```

```
## [1] "Flag"                               "Count"
## [3] "Service"                            "Dst_host_same_src_port_rate"
## [5] "Dst_host_diff_srv_rate"             "Hot"
## [7] "Dst_host_count"                     "Dst_host_srv_count"
## [9] "Protocol_type"                      "Wrong_fragment"
## [11] "Dst_host_serror_rate"               "Dst_host_srv_diff_host_rate"
## [13] "Dst_host_rerror_rate"               "Srv_rerror_rate"
## [15] "Srv_diff_host_rate"                 "Duration"
## [17] "Same_srv_rate"                      "Dst_host_same_srv_rate"
## [19] "Logged_in"                          "Diff_srv_rate"
## [21] "Rerror_rate"                        "Srv_error_rate"
## [23] "Land"                               "Num_file_creations"
## [25] "Is_guest_login"                     "Serror_rate"
## [27] "Dst_host_srv_serror_rate"           "Dst_host_srv_rerror_rate"
## [29] "Root_shell"                         "Urgent"
## [31] "Num_ccess_files"                    "Num_shells"
## [33] "Num_root"                           "Is_host_login"
## [35] "Num_compromised"                   "Num_failed_logins"
## [37] "Su_attempted"                       "Dst_bytes"
## [39] "Src_bytes"                          "Srv_count"
```

Veamos si con las primeras dos variables se puede determinar gráficamente alguna separación notable con respecto al tráfico *normal* o a los *ataques*. Para ello primero debemos cargar el conjunto de datos de entrenamiento y eliminar las columnas de etiquetas que no se utilizarán.

```
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv",
                           sep = ",", header = TRUE)

#Eliminando características innecesarias
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL
```

Adicionalmente se asegurará que las columnas variables predictoras sean de tipo *numérico*.

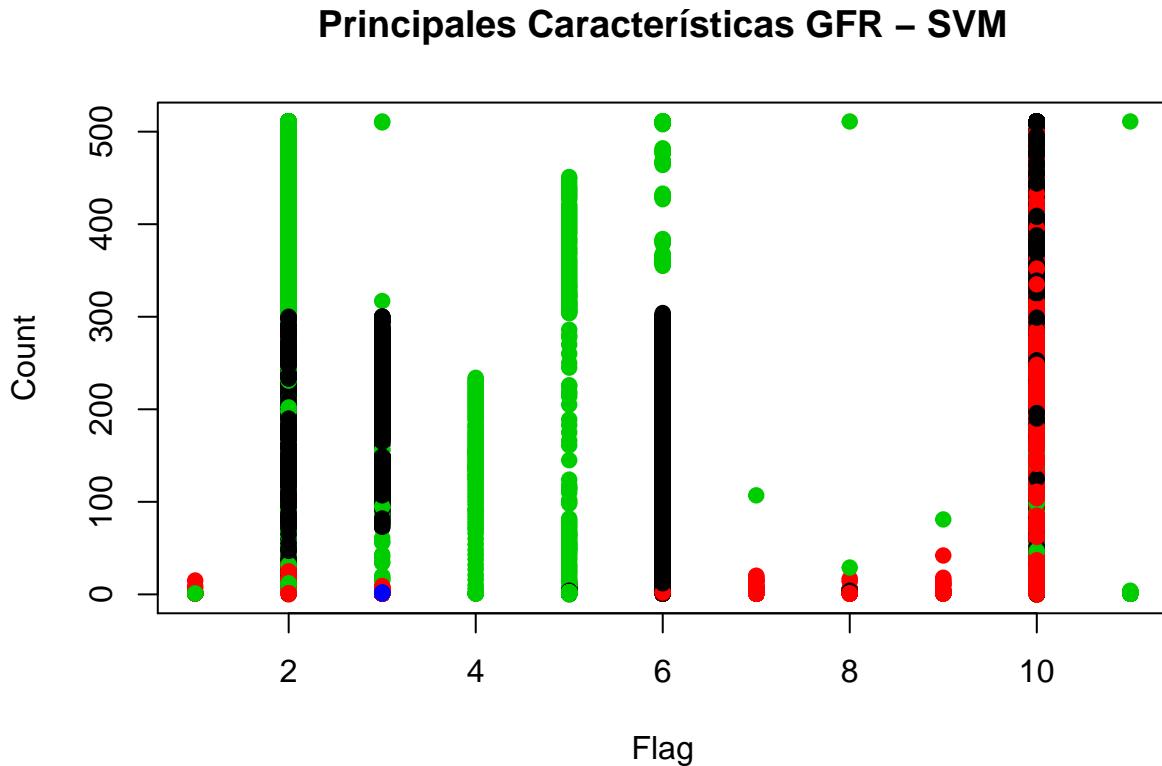
```
for (i in 1:(ncol(dataset)-1))
  dataset[,i] = as.numeric(dataset[,i])
```

Luego es necesario crear un vector de colores para poder diferenciar entre las diferentes clases en el gráfico.

```
colors = as.character(dataset[, ncol(dataset)])
colors[colors == "normal"] = "black"
colors[colors == "DoS"] = "red"
colors[colors == "Probing"] = "green"
colors[colors == "R2L"] = "blue"
colors[colors == "U2R"] = "cyan"
```

Donde *negro* corresponde a la clase normal, *rojo* a ataques DoS, *verde* a ataques Probing, *azul* a ataques *R2L* y *cian* a ataques *U2R*. El gráfico de las dos primeras características más importantes se muestra a continuación.

```
par(mfrow = c(1,1))
plot(dataset[, rownames(svm.gfr)[1]], dataset[, rownames(svm.gfr)[2]],
      col = dataset$Label, pch = 19,
      xlab = "Flag", ylab = "Count",
      main = "Principales Características GFR – SVM")
```



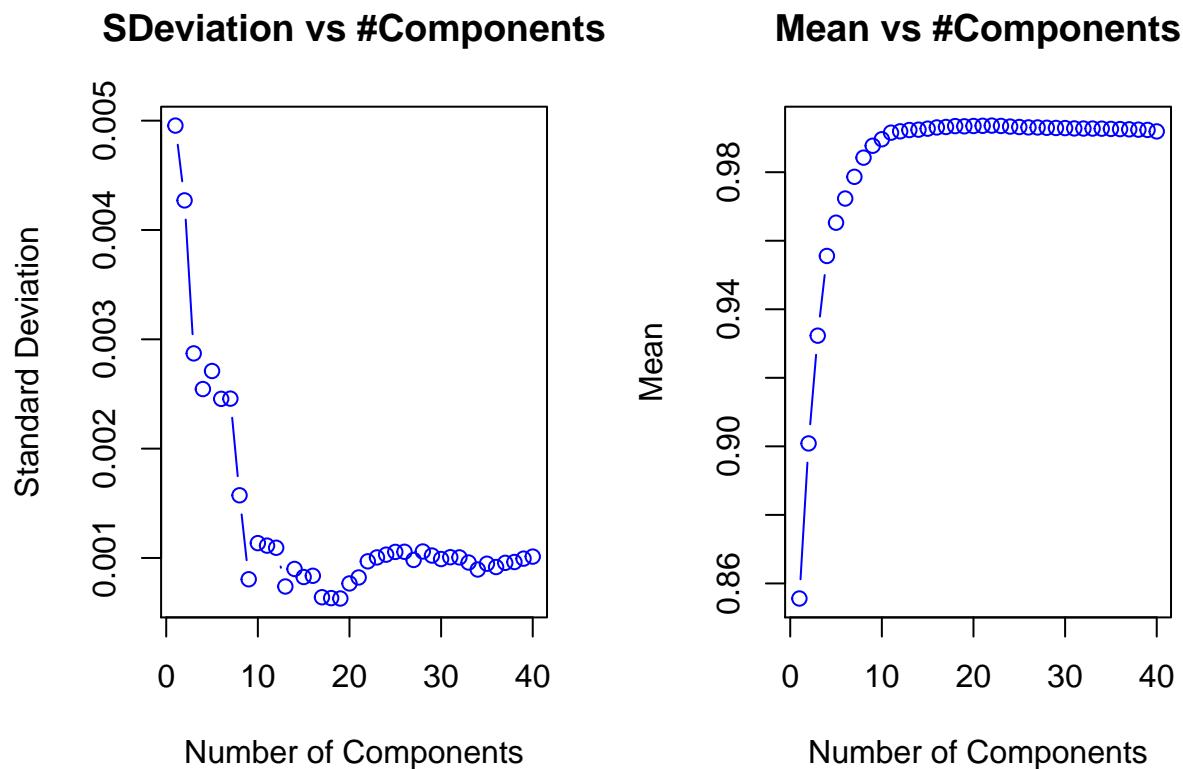
En la gráfica anterior se pueden observar patrones con respecto a los valores de las características que pueden ayudar a la separación de las clases, por ejemplo, si *Flag* = 2 y *Count*  $\geq 300$ , se puede decir que dicho registro pertenece a un ataque probing. Sin embargo, SVM no funciona de esta manera (estableciendo reglas), a su vez, funciona delimitando fronteras, esto puede presentar una situación a analizar para la decisión del uso de SVM o de NN por ejemplo. Ya que NN si es capaz de establecer reglas como las mencionadas anteriormente.

Ahora se procede a realizar los pasos para el análisis de las características a elegir de forma definitiva. Inicialmente, se calculan y la *media* y *desviación estándar*.

```
mean.values = apply(svm.gfr, 1, mean)
sdeviation.values = apply(svm.gfr, 1, sd)
```

Y se grafican las medidas calculadas previamente.

```
par(mfrow = c(1,2))
plot(sdeviation.values[2:length(mean.values)],
      type = "b", col = "blue",
      main = "SDeviation vs #Components",
      xlab = "Number of Components", ylab = "Standard Deviation")
plot(mean.values[2:length(mean.values)],
      type = "b", col = "blue",
      main = "Mean vs #Components",
      xlab = "Number of Components", ylab = "Mean")
```



Para la selección de características en esta sección se usa un criterio similar al de *codo de jambu*. Acá se observó que la articulación se logra con 9 o 10 características. Como se observa en el gráfico de la *desviación estándar*, los resultados con 9 variables son más estables que con 10 variables y por dicho motivo se seleccionará dicho número como cantidad de variables ideal para el modelo de SVM. Las mismas se listan a continuación.

```
rownames(svm.gfr)[1:9]
```

```
## [1] "Flag"                               "Count"
## [3] "Service"                            "Dst_host_same_src_port_rate"
## [5] "Dst_host_diff_srv_rate"             "Hot"
## [7] "Dst_host_count"                     "Dst_host_srv_count"
## [9] "Protocol_type"
```

Luego, se puede observar con la selección de 9 variables se logaría reducir la dimensionalidad del conjunto de datos en un 77.5%.

## Efecto de GFR sobre NN

En esta sección se evalua el efecto de la aplicación de GFR sobre SVM. Para esto se entrena modelos haciendo uso de [1,40] variables predictoras y se calculan las tasa de aciertos en cada iteración. Para validar el modelo se hará uso de la técnica de *validación cruzada de 10 conjuntos*. Esta sección está dividida en dos partes: *entrenamiento y análisis*.

### Entrenamiento

Acá se describen todas las tareas realizadas en el proceso de entrenamiento de los modelos en el rango [1,40] variables predictoras. Se iniciará preparando el ambiente de trabajo: limpiar variables parciales, cargar paquetes necesarios y archivos de funciones.

```
rm(list = ls())
require("nnet")
source("functions/functions.R")
```

Se eliminan las etiquetas que no serán usadas.

```
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL
```

Se transforman las variables predictoras a tipo *numérico*.

```
for (i in 1:(ncol(dataset)-1))
  dataset[,i] = as.numeric(dataset[,i])
```

Se transforma la variable objetivo a tipo factor, esto debido a que se realizarán labores de *clasificación*. Es importante recordar que la variable objetivo posee cinco clases: DoS, normal, Probing, R2L y U2R.

```
dataset[,ncol(dataset)] = as.factor(dataset[,ncol(dataset)])
```

Se escala el conjunto de datos para que todas las variables predictoras tengan *media* cero (0) y *desviación estándar* uno (1).

```
dataset = ScaleSet(dataset)
```

En este punto se tiene todo listo para aplicar el algoritmo GFR, este fue comprimido en la función **GFR** que recibe como parámetros un *dataframe* y el tipo de algoritmo que se desea usar **SVM** o **NN**,

```
results = GFR(dataset, "SVM")
```

GFR retorna una matriz de dimensiones 41x10, donde 41 corresponde a la cantidad de resultados por característica eliminada y 10 son los resultados obtenidos por iteración durante el proceso de *validación cruzada de 10 conjuntos*. Por último, la matriz será almacenada en un objeto para su posterior análisis. Esto debido a que este proceso duró 12 días para su culminación.

```
saveRDS(results, " ../source/feature_selection/NN/results_GFR.rds")
```

## Análisis

Esta sección describe las actividades realizadas para la fase de análisis. Estas iniciarán limpiando el ambiente de trabajo de variables parciales y cargando el archivo de funciones.

```
rm(list = ls())
source("../source/functions/functions.R")
```

A continuación se cargará el objeto con los resultados obtenidos en la sección anterior.

```
nn.gfr = readRDS("../source/feature_selection/NN//results_GFR.rds")
```

A continuación se ilustran los resultados de las características obtenidas en la sección anterior. Las mismas están ordenadas de manera descendente; es decir, las primeras posiciones representan las más importantes.

```
rownames(nn.gfr)[-nrow(nn.gfr)]
```

```
## [1] "Count"                                "Protocol_type"
## [3] "Dst_host_srv_count"                     "Dst_host_same_src_port_rate"
## [5] "Dst_host_error_rate"                    "Dst_host_count"
## [7] "Hot"                                    "Dst_host_error_rate"
## [9] "Wrong_fragment"                         "Service"
## [11] "Logged_in"                             "Is_guest_login"
## [13] "Dst_host_diff_srv_rate"                "Srv_count"
## [15] "Num_failed_logins"                     "Error_rate"
## [17] "Dst_host_same_srv_rate"                "Num_compromised"
## [19] "Flag"                                   "Num_access_files"
## [21] "Is_host_login"                         "Dst_bytes"
## [23] "Dst_host_srv_error_rate"               "Diff_srv_rate"
## [25] "Srv_error_rate"                        "Duration"
## [27] "Srv_diff_host_rate"                   "Num_file_creations"
## [29] "Num_root"                             "Dst_host_srv_error_rate"
## [31] "Dst_host_srv_diff_host_rate"           "Error_rate"
## [33] "Srv_error_rate"                        "Num_shells"
## [35] "Root_shell"                            "Land"
## [37] "Same_srv_rate"                         "Src_bytes"
## [39] "Su_attempted"                          "Urgent"
```

Veamos si con las primeras dos variables más importantes se puede determinar gráficamente alguna separación notable con respecto al *trafico normal* o a los *ataques*. Para ello primero debemos cargar el conjunto de datos de entrenamiento y eliminar las columnas de etiquetas que no se utilizarán.

```
dataset.training = read.csv("../dataset/NSLKDD_Training_New.csv",
                           sep = ",", header = TRUE)
#Eliminando características innecesarias
dataset = dataset.training
dataset$Label_Normal_TypeAttack = NULL
dataset$Label_Num_Classifiers = NULL
dataset$Label_Normal_or_Attack = NULL
```

Adicionalmente se asegurará que las columnas de variables predictoras sean de tipo *numérico*.

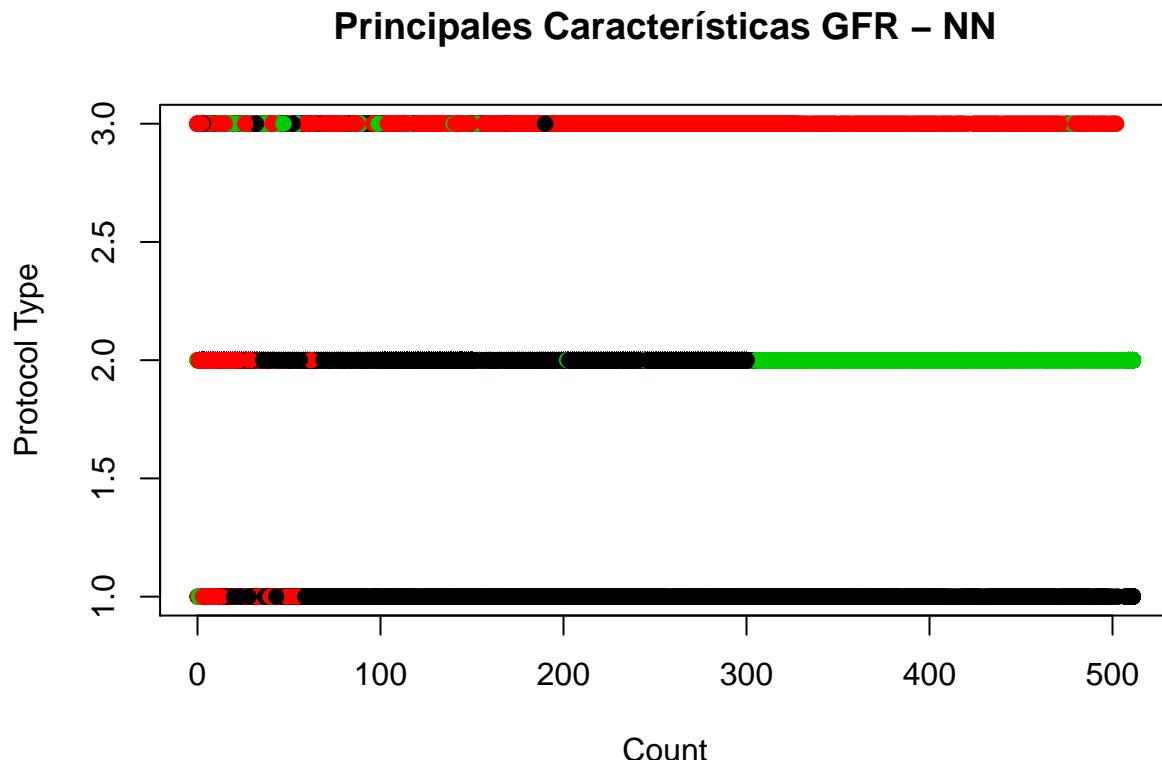
```
for (i in 1:(ncol(dataset)-1))
  dataset[,i] = as.numeric(dataset[,i])
```

Luego, es necesario crear un vector de colores para poder diferenciar las diferentes clases en el gráfico.

```
colors = as.character(dataset[,ncol(dataset)])
colors[colors == "normal"] = "black"
colors[colors == "DoS"] = "red"
colors[colors == "Probing"] = "green"
colors[colors == "R2L"] = "blue"
colors[colors == "U2R"] = "cyan"
```

Donde *negro* corresponde a la clase normal, *rojo* a ataques DoS, *verde* a ataques Probing, *azul* a ataques R2L y *cian* a ataques U2R. El gráfico de las dos primeras características más importantes se muestra a continuación.

```
par(mfrow = c(1,1))
plot(dataset[, rownames(nn.gfr)[1]], dataset[, rownames(nn.gfr)[2]],
  col = dataset$Label, pch = 19,
  xlab = "Count", ylab = "Protocol Type",
  main = "Principales Características GFR - NN")
```



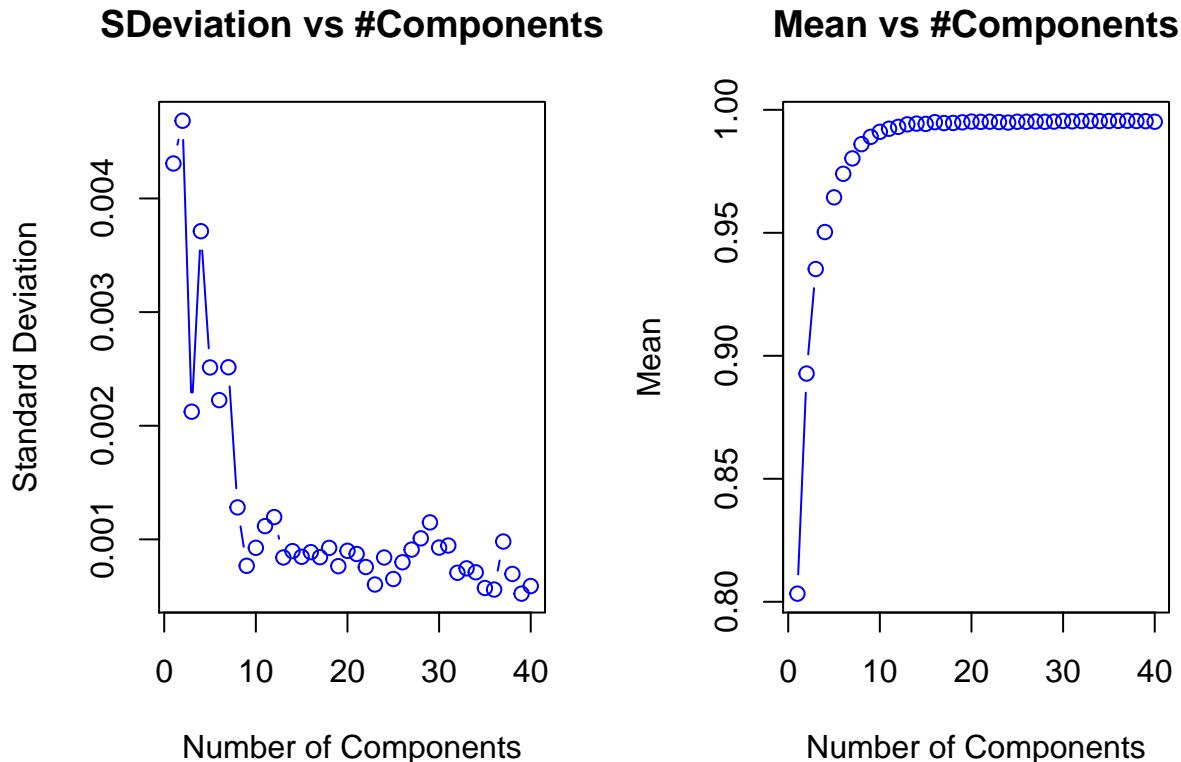
En el gráfico se observan patrones que pueden separar notablemente a las diferentes clases. Por ejemplo, si Count > 50 y Protocol Type = 1, entonces el tráfico es normal. El gráfico muestra separaciones bastante ordenadas y esto ayuda mucho a la forma en la que se comportan las redes neuronales, ya que este algoritmo busca patrones analíticos en los datos que ayuden a la clasificación.

Ahora se procede a realizar los pasos para el análisis de las características a elegir de forma definitiva. inicialmente, se calculan la *media* y *desviación estándar*.

```
mean.values = apply(nn.gfr, 1, mean)
sdeviation.values = apply(nn.gfr, 1, sd)
```

Y se grancian las medidas calculadas previamente.

```
par(mfrow = c(1,2))
plot(sdeviation.values[2:length(mean.values)],
      type = "b", col = "blue",
      main = "SDDeviation vs #Components",
      xlab = "Number of Components", ylab = "Standard Deviation")
plot(mean.values[2:length(mean.values)],
      type = "b", col = "blue",
      main = "Mean vs #Components",
      xlab = "Number of Components", ylab = "Mean")
```



Para la selección de características en esta sección se usa un criterio similar al de *codo de jambu*. Acá se pbserva que la articulación se logra con 9 ó 10 características. En el gráfico de *desviación estándar*, los resultados con 9 variables son más estables que con 10 variables y por dicho motivo se seleccionará dicho número como cantidad de variables ideal para el modelo de NN. Las mismas se listan a continuación.

```
rownames(nn.gfr) [1:9]
```

```
## [1] "Count"                      "Protocol_type"
## [3] "Dst_host_srv_count"          "Dst_host_same_src_port_rate"
```

```

## [5] "Dst_host_error_rate"
## [7] "Hot"
## [9] "Wrong_fragment"

```

Luego, se observa que con la selección de 9 variables se logra reducir la dimensionalidad del conjunto de datos en un 77.5%.

## Conclusión

Con GFR se muestra que se puede obtener buenos resultados para ambos modelos (SVM y NN) usando el 22.5% de las variables totales. Se observó también que la desviación estándar de los resultados en ambos casos es pequeña, por no decir mínima, y esto es un hecho que respalda positivamente a los resultados obtenidos, indicando que la reducción de dimensionalidad para este escenario es válido y efectivo. Como aspecto a destacar está el orden de las características seleccionadas es diferente con respecto a los modelos de SVM y NN y esto tiene su naturaleza en que el algoritmo de SVM tiene una orientación descriptiva y busca aquellas características que posicionalmente dividan mejor a las clases en un espacio geométrico *N-dimensional*. Por otra parte, el algoritmo de NN tiene un enfoque analítico que busca patrones en las diferentes características que sirvan como premisa para poder separar las diferentes clases. Es por esto, que se obtuvieron diferentes resultados en los diferentes modelos. Sin embargo, se puede destacar de acá que la unión de las primeras 9 características seleccionadas para ambos modelos son fuertes candidatas a variables predictoras a tomar en cuenta para la detección de intrusos en redes de computadoras.

## Análisis entre PCA y GFR

Una vez realizadas las pruebas concernientes se puede observar que desde un punto de vista de rendimiento, PCA alcanza el 99% de precisión con 7 variables mientras que GFR alcanza 99% con 9 variables. Adicionalmente, PCA tardó a penas 13 horas para poder realizar la selección de componentes principales, mientras que con GFR el proceso duró 12 días. Sin embargo, con GFR se pudo mantener el nombre de las variables importantes y no se perdió interpretación como si sucedió con la aplicación de PCA.