Solución 1

Detección de intrusos en redes

Estos datos fueron usados para la edición de 1999 del KDD cup. Los datos fueron generados por Lincoln Labs: *Nueve semanas de registro de paquetes TCP fueron recolectadas para una red LAN de una oficina de las fuerzas aéreas de USA.* Durante el uso de la LAN, *varios ataques* fueron ejecutados por el personal. El paquete crudo fue agregado junto con la información de la conexión.

Para cada registro, algunas características extra fueron derivadas, basados en conocimiento del dominio sobre ataques a redes; *hay 38 tipos diferentes de ataques, pertenecientes a 4 categorías principales*. Algunos tipos de ataque aparecen solo en los datos de prueba(test data), y las frecuencias de los tipo de ataque en los conjuntos de entrenamiento y prueba no son las mismas(para hacerlo más realista). Información adicional sobre los datos puede ser encontrada en (http://kdd.ics.uci.edu/databases/kddcup99/task.html (http://kdd.ics.uci.edu/databases/kddcup99/task.html)) y los resumenes de los resultados de la competencia KDD cup (http://cseweb.ucsd.edu/~elkan/clresults.html (http://cseweb.ucsd.edu/~elkan/clresults.html)). En la última página también se indica que hay una matriz de costo asociada con las equivocaciones. El ganador de la competencia usó árboles de decisión C5 en combinación con boosting y bagging.

Referencias:

- PNrule: *A New Framework for Learning Classifier Models in Data Mining (A Case-Study in Network Intrusion Detection) (2000) by R. Agarwal and M. V. Joshi*. This paper proposes a new, very simple rule learning algorithm, and tests it on the network intrusion dataset. In the first stage, rules are learned to identify the target class, and then in the second stage, rules are learned to identify cases that were incorrectly classified as positive according to the first rules.

Pasos a llevar a cabo en la solución 1

- Cargar las librerías a utilizar
- Cargar los datos e importarlos a un dataframe
- Visualizar los datos
- Limpiar y transformar los datos
- Códificar los datos
- Seleccionar los parámetros más importantes
- Separando el conjunto de datos de entrenamiento y de validación
- Selección de algoritmos y métodos
- Validación Cruzada con KFold
- Resumen de los métodos utilizados
- Comparación de resultados con el ganador del KDDCup

Cargar las librerías a utilizar

In [1]:
```
%matplotlib inline

from time import time
from itertools import product
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import graphviz
from sklearn import metrics
from sklearn.metrics import make_scorer, accuracy_score, confusion_matrix
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import SelectFromModel
from sklearn.feature_selection import chi2

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.tree import export_graphviz
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier

from IPython.display import display, HTML

# Modulo personal para hacer mas claro el ejercicio
from egironML import EDA
from egironML import run_kfold

plt.style.use('seaborn-white')
```

Cargar los datos e importarlos a un dataframe

In [2]:
```
# Cargamos los datos de Detección de intrusos por medio de un modelo personalizad
o
ataques_10perc, ataques_correg_test_10perc = EDA.load_attack_rawData()
```

```
Cantidad de observaciones (Entrenamiento):  494021
Cantidad de observaciones (Validación):  311029
Cantidad de ataques (Entrenamiento):  23
Cantidad de ataques (Validación):  38
```

Visualizar los datos

In [3]:
```
ataques_10perc.head()
```

Out[3]:

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...
0	0	tcp	http	SF	181	5450	0	0	0	0	...
1	0	tcp	http	SF	239	486	0	0	0	0	...
2	0	tcp	http	SF	235	1337	0	0	0	0	...
3	0	tcp	http	SF	219	1337	0	0	0	0	...
4	0	tcp	http	SF	217	2032	0	0	0	0	...

5 rows × 42 columns

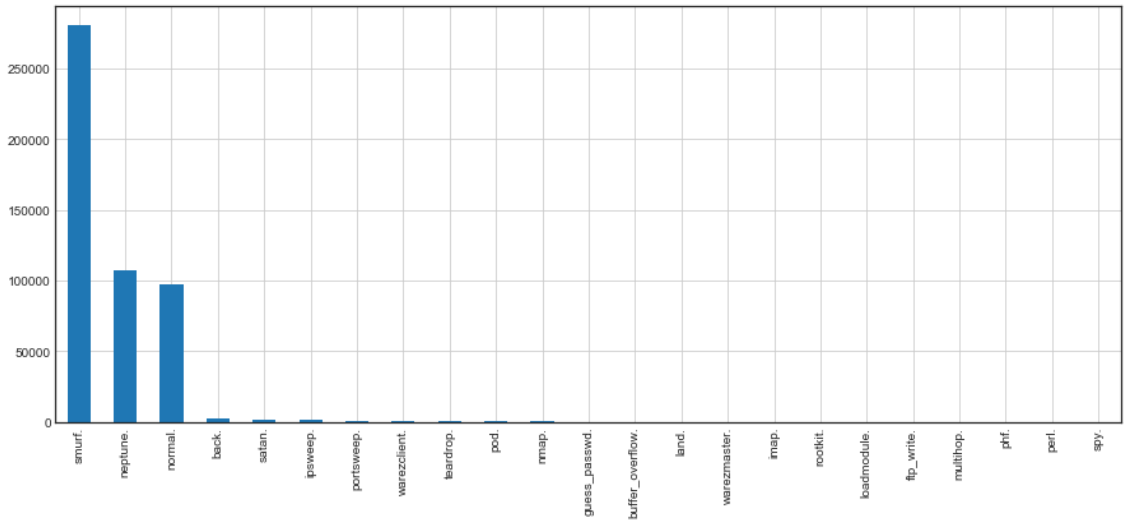
In [4]:
```
EDA.printall(ataques_correg_test_10perc, 15)
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent
0	0	udp	private	SF	105	146	0	0	0
1	0	udp	private	SF	105	146	0	0	0
2	0	udp	private	SF	105	146	0	0	0
3	0	udp	private	SF	105	146	0	0	0
4	0	udp	private	SF	105	146	0	0	0
5	0	udp	private	SF	105	146	0	0	0
6	0	udp	domain_u	SF	29	0	0	0	0
...	...	...	...	...	...	...	...	...	...
311022	0	udp	private	SF	105	105	0	0	0
311023	0	udp	private	SF	105	105	0	0	0
311024	0	udp	private	SF	105	147	0	0	0
311025	0	udp	private	SF	105	147	0	0	0
311026	0	udp	private	SF	105	147	0	0	0
311027	0	udp	private	SF	105	147	0	0	0
311028	0	udp	private	SF	105	147	0	0	0

In [5]:
```
# ataques_correg_test_10perc.sample(3)
```

In [6]:
```
# Veamos la distribucción de los ataques
ataques_10perc.attack_types.value_counts().plot(kind='bar', grid=True, figsize=(15, 6))
```

Out[6]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1220b080>

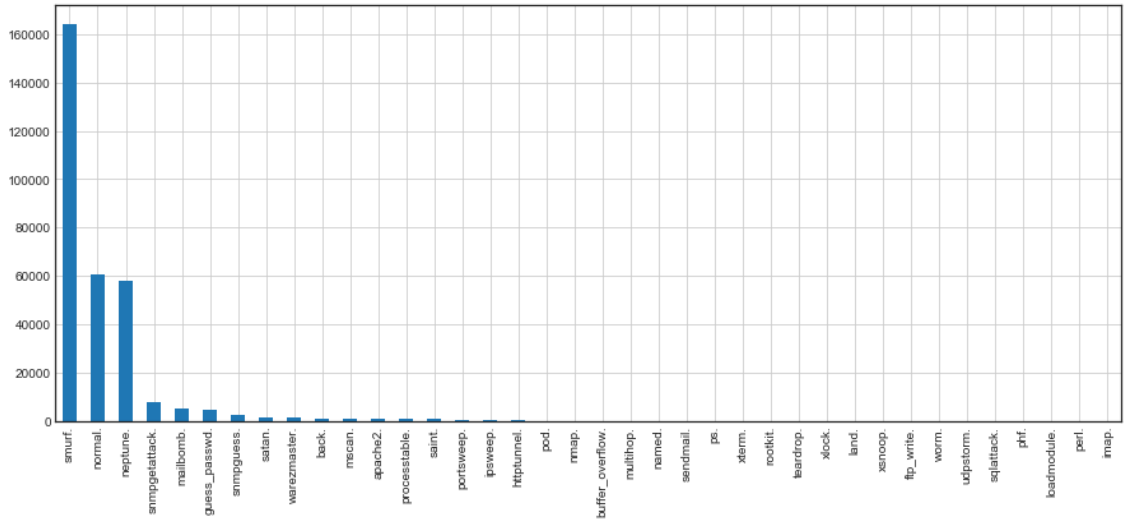


In [7]:
```
ataques_10perc.attack_types.value_counts()
```

Out[7]:
```
smurf.              280790
neptune.            107201
normal.              97278
back.                 2203
satan.                1589
ipsweep.              1247
portsweep.            1040
warezclient.          1020
teardrop.              979
pod.                   264
nmap.                  231
guess_passwd.           53
buffer_overflow.        30
land.                   21
warezmaster.            20
imap.                   12
rootkit.                10
loadmodule.              9
ftp_write.               8
multihop.                7
phf.                     4
perl.                    3
spy.                     2
Name: attack_types, dtype: int64
```

In [8]:
```
ataques_correg_test_10perc.attack_types.value_counts().plot(kind='bar', grid=True
, figsize=(15, 6))
```

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1220b9e8>



In [9]:
```
ataques_correg_test_10perc.attack_types.value_counts()
```

Out[9]:
```
smurf.              164091
normal.              60593
neptune.             58001
snmpgetattack.        7741
mailbomb.             5000
guess_passwd.         4367
snmpguess.            2406
satan.                1633
warezmaster.          1602
back.                 1098
mscan.                1053
apache2.               794
processtable.          759
saint.                 736
portsweep.             354
ipsweep.               306
httptunnel.            158
pod.                    87
nmap.                   84
buffer_overflow.        22
multihop.               18
named.                  17
sendmail.               17
ps.                     16
xterm.                  13
rootkit.                13
teardrop.               12
xlock.                   9
land.                    9
xsnoop.                  4
ftp_write.               3
worm.                    2
udpstorm.                2
sqlattack.               2
phf.                     2
loadmodule.              2
perl.                    2
imap.                    1
Name: attack_types, dtype: int64
```

In [10]:
```
print("Cantidad de ataques (Entrenamiento): ",len(ataques_10perc.attack_types.uni
que()))
print("Cantidad de ataques (Validación): ",len(ataques_correg_test_10perc.attack_
types.unique()))
```

```
Cantidad de ataques (Entrenamiento):  23
Cantidad de ataques (Validación):  38
```

Limpiar y transformar los datos

Como se aprecia en los datos anteriores, la cantidad de ataques son diferentes en en el conjunto de datos de entrenamiento y valdación. Igualmente poseen un punto al final de cada atributo de tipo de ataque. Por ultimo hay algunas variables o predictores de tipo categórico.

In [11]:
```
# Reemplazamos el . en los vlores del precitor de tipos de ataques.
EDA.replace_column_string(ataques_10perc, 'attack_types', '.', '')
EDA.replace_column_string(ataques_correg_test_10perc, 'attack_types', '.', '')
```

In [12]:
```
ataques_10perc.attack_types.unique()
```

Out[12]:
```
array(['normal', 'buffer_overflow', 'loadmodule', 'perl', 'neptune',
       'smurf', 'guess_passwd', 'pod', 'teardrop', 'portsweep', 'ipsweep',
       'land', 'ftp_write', 'back', 'imap', 'satan', 'phf', 'nmap',
       'multihop', 'warezmaster', 'warezclient', 'spy', 'rootkit'], dtype=object
)
```

In [13]:
```
ataques_correg_test_10perc.attack_types.unique()
```

Out[13]:
```
array(['normal', 'snmpgetattack', 'named', 'xlock', 'smurf', 'ipsweep',
       'multihop', 'xsnoop', 'sendmail', 'guess_passwd', 'saint',
       'buffer_overflow', 'portsweep', 'pod', 'apache2', 'phf', 'udpstorm',
       'warezmaster', 'perl', 'satan', 'xterm', 'mscan', 'processtable',
       'ps', 'nmap', 'rootkit', 'neptune', 'loadmodule', 'imap', 'back',
       'httptunnel', 'worm', 'mailbomb', 'ftp_write', 'teardrop', 'land',
       'sqlattack', 'snmpguess'], dtype=object)
```

In [14]:
```
# Verificamos el tipo de dato en cada atributo o predictor
ataques_10perc.info()
ataques_correg_test_10perc.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 494021 entries, 0 to 494020
Data columns (total 42 columns):
duration                        494021 non-null int64
protocol_type                   494021 non-null object
service                         494021 non-null object
flag                            494021 non-null object
src_bytes                       494021 non-null int64
dst_bytes                       494021 non-null int64
land                            494021 non-null int64
wrong_fragment                  494021 non-null int64
urgent                          494021 non-null int64
hot                             494021 non-null int64
num_failed_logins               494021 non-null int64
logged_in                       494021 non-null int64
num_compromised                 494021 non-null int64
root_shell                      494021 non-null int64
su_attempted                    494021 non-null int64
num_root                        494021 non-null int64
num_file_creations              494021 non-null int64
num_shells                      494021 non-null int64
num_access_files                494021 non-null int64
num_outbound_cmds               494021 non-null int64
is_host_login                   494021 non-null int64
is_guest_login                  494021 non-null int64
count                           494021 non-null int64
srv_count                       494021 non-null int64
serror_rate                     494021 non-null float64
srv_serror_rate                 494021 non-null float64
rerror_rate                     494021 non-null float64
srv_rerror_rate                 494021 non-null float64
same_srv_rate                   494021 non-null float64
diff_srv_rate                   494021 non-null float64
srv_diff_host_rate              494021 non-null float64
dst_host_count                  494021 non-null int64
dst_host_srv_count              494021 non-null int64
dst_host_same_srv_rate          494021 non-null float64
dst_host_diff_srv_rate          494021 non-null float64
dst_host_same_src_port_rate     494021 non-null float64
dst_host_srv_diff_host_rate     494021 non-null float64
dst_host_serror_rate            494021 non-null float64
dst_host_srv_serror_rate        494021 non-null float64
dst_host_rerror_rate            494021 non-null float64
dst_host_srv_rerror_rate        494021 non-null float64
attack_types                    494021 non-null object
dtypes: float64(15), int64(23), object(4)
memory usage: 158.3+ MB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 311029 entries, 0 to 311028
Data columns (total 42 columns):
duration                        311029 non-null int64
protocol_type                   311029 non-null object
service                         311029 non-null object
flag                            311029 non-null object
src_bytes                       311029 non-null int64
dst_bytes                       311029 non-null int64
land                            311029 non-null int64
wrong_fragment                  311029 non-null int64
urgent                          311029 non-null int64
hot                             311029 non-null int64
num_failed_logins               311029 non-null int64
logged_in                       311029 non-null int64
num_compromised                 311029 non-null int64
root_shell                      311029 non-null int64
```

Como se puede apreciar el conjunto de entrenamiento tiene **494021** observaciones y el de validación **311029**; todas las vairables estan completas por lo que no hay valores faltantes o nulos. De igual forma se notan 4 variables categóricas (*protocol_type, service, flag y attack_types*).

In [15]:
```
# Varificamos de nuevo o de otra forma si hay datos faltantes o nulos
# ataques_10perc.isnull().sum()
EDA.hasNull(ataques_10perc)
```

```
duration                        0
protocol_type                   0
service                         0
flag                            0
src_bytes                       0
dst_bytes                       0
land                            0
wrong_fragment                  0
urgent                          0
hot                             0
num_failed_logins               0
logged_in                       0
num_compromised                 0
root_shell                      0
su_attempted                    0
num_root                        0
num_file_creations              0
num_shells                      0
num_access_files                0
num_outbound_cmds               0
is_host_login                   0
is_guest_login                  0
count                           0
srv_count                       0
serror_rate                     0
srv_serror_rate                 0
rerror_rate                     0
srv_rerror_rate                 0
same_srv_rate                   0
diff_srv_rate                   0
srv_diff_host_rate              0
dst_host_count                  0
dst_host_srv_count              0
dst_host_same_srv_rate          0
dst_host_diff_srv_rate          0
dst_host_same_src_port_rate     0
dst_host_srv_diff_host_rate     0
dst_host_serror_rate            0
dst_host_srv_serror_rate        0
dst_host_rerror_rate            0
dst_host_srv_rerror_rate        0
attack_types                    0
dtype: int64
```

In [16]:
```
ataques_10perc.isin([np.nan, np.inf, -np.inf]).any(1).sum()
```

Out[16]: 0

In [76]:
```
#ataques_correg_test_10perc.apply(EDA.hasNull)
```

In [19]:
```
ataques_correg_test_10percAll = ataques_correg_test_10perc.copy() # Salvamos una
copia con todos los datos para validar
```

In [20]:
```
len(ataques_correg_test_10percAll)
```

Out[20]:
```
311029
```

In [21]:
```
# Eliminar duplicados
EDA.delduplicates(ataques_10perc)
EDA.delduplicates(ataques_correg_test_10perc)
```

```
Eliminó 145586 observaciones de un total de 494021
Eliminó 77291 observaciones de un total de 311029
```

Es necesario tener en cuenta que los resultados que se muestran en las matríz de confusión del ganador del KDD Cup son con todos los 311029 datos incluyendo los duplicados.

In [22]:
```
# Ahora transformamos algunos datos como lo son los tipo de ataques
# Agrupamos los tipos de ataques en las 4 categorías recomendadas
EDA.create_category_attack(ataques_10perc)
EDA.create_category_attack(ataques_correg_test_10perc)
EDA.create_category_attack(ataques_correg_test_10percAll)
```

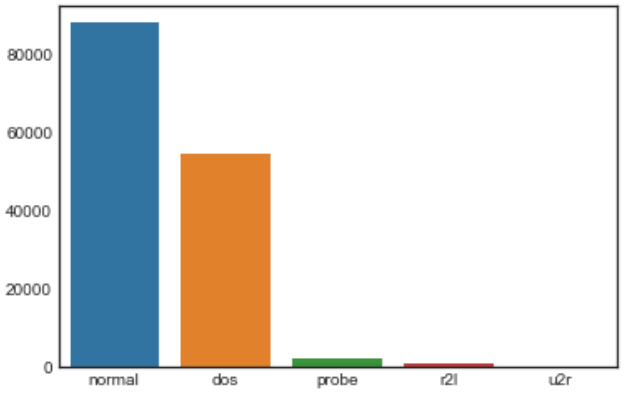
```
Categorías encontradas: ['normal' 'u2r' 'dos' 'r2l' 'probe']
Categorías encontradas: ['normal' 'unknown' 'dos' 'probe' 'r2l' 'u2r']
Categorías encontradas: ['normal' 'unknown' 'dos' 'probe' 'r2l' 'u2r']
```

In [23]:
```
# Ahora creamos una variable binaria (0/1 o No/Yes o good/bad) para guardar las c
onexiones con o sin ataques
# Esto nos permite aplicar modelos de regresión mas adelante
EDA.create_category_binAttack(ataques_10perc)
EDA.create_category_binAttack(ataques_correg_test_10perc)
EDA.create_category_binAttack(ataques_correg_test_10percAll)
```

```
Categorías encontradas: [0 1]
Categorías encontradas: [0 1]
Categorías encontradas: [0 1]
```

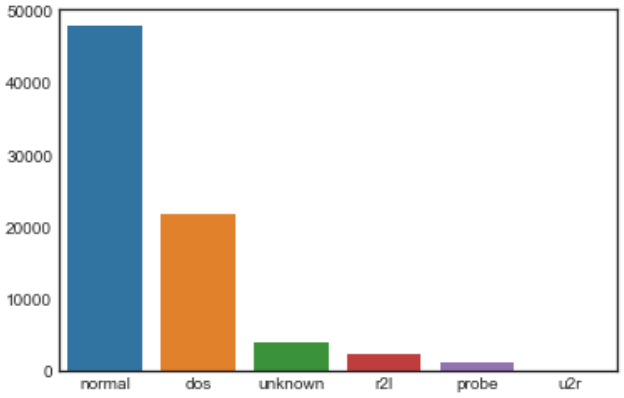
In [24]:
```
gdata = ataques_10perc.attack_category.value_counts()
sns.barplot(x=gdata.keys().tolist(), y=gdata.data.tolist())
```

Out[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a121db4a8>
```

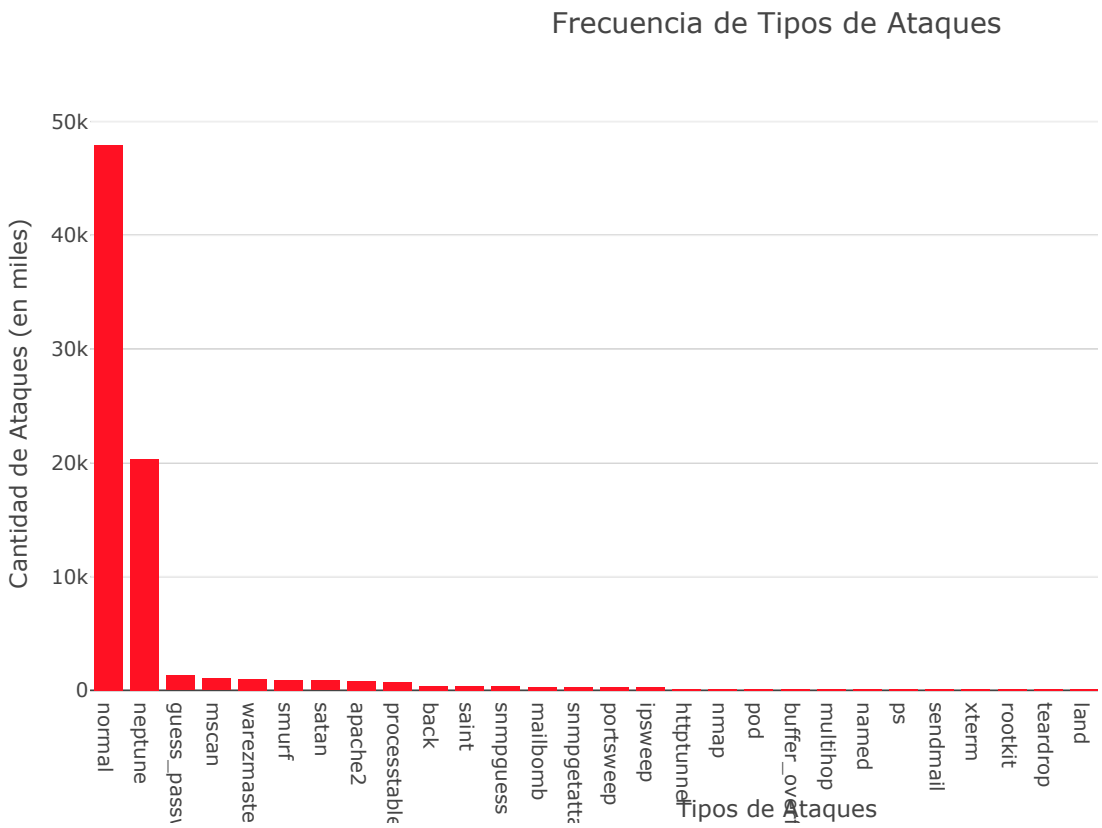


In [25]:
```
gdata = ataques_correg_test_10perc.attack_category.value_counts()
sns.barplot(x=gdata.keys().tolist(), y=gdata.data.tolist())
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1a0ff668>



In [26]:
```
import plotly
from plotly.graph_objs import Bar, Scatter, Layout, Figure
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
plotly.offline.init_notebook_mode(connected=True)

gdata = ataques_correg_test_10perc.attack_types.value_counts()
attackctg = Bar(x=gdata.keys().tolist(), y=gdata.data.tolist(), name='Attack', ma
rker=dict(color='#ff1123') )

data = [attackctg]
layout = Layout(title="Frecuencia de Tipos de Ataques", xaxis=dict(title='Tipos d
e Ataques'),
                yaxis=dict(title='Cantidad de Ataques (en miles)'))
fig = Figure(data=data, layout=layout)

iplot(fig)
```

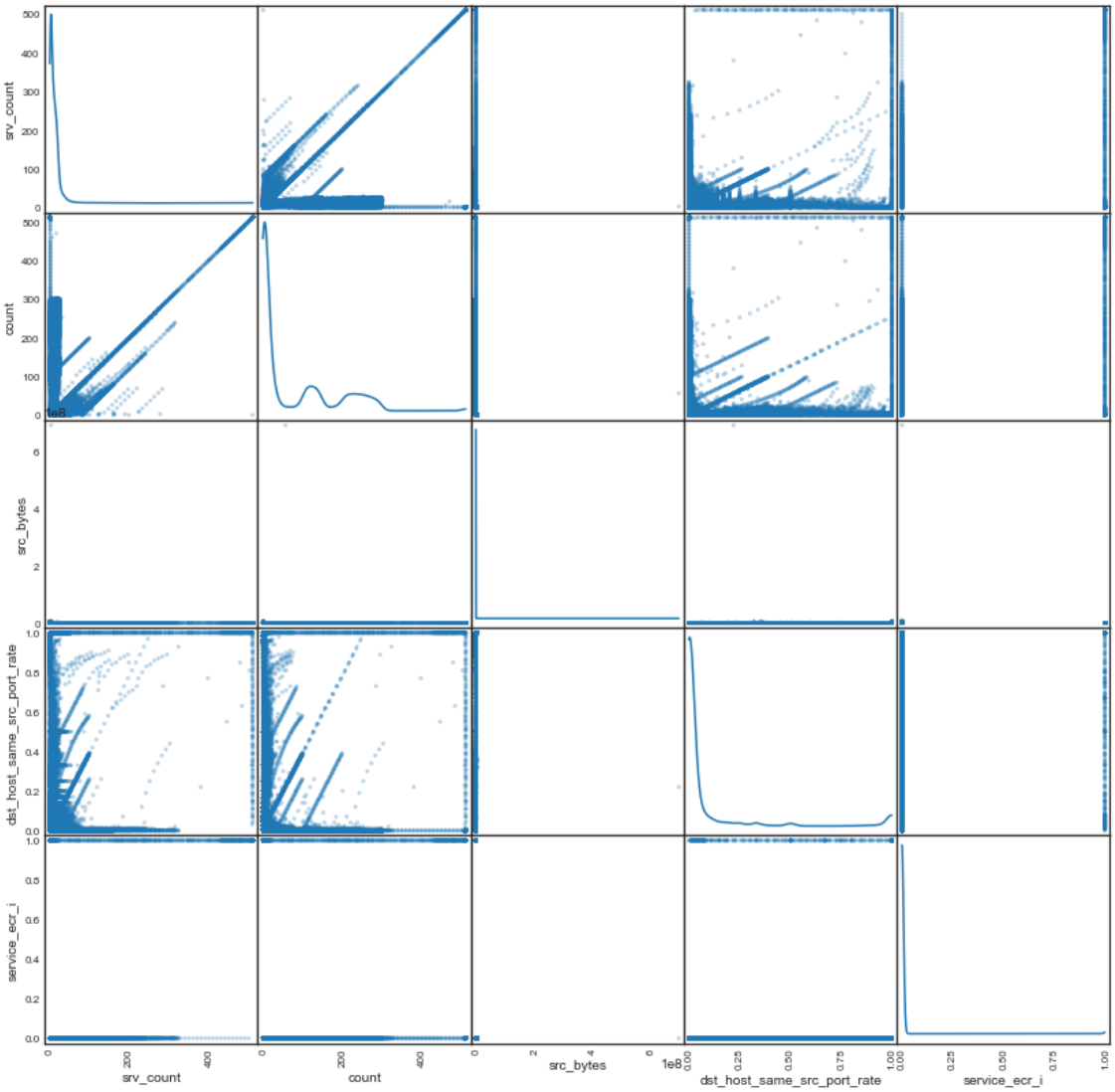


In [27]:
```
corr = ataques_10perc.corr()
f, ax = plt.subplots(figsize=(12, 12))
sns.heatmap(corr, mask=np.zeros_like(corr, dtype=np.bool), cmap=sns.diverging_pal
ette(220, 10, as_cmap=True),
            square=False, ax=ax)
```

Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1a9cad68>

In [77]:
```
# Esto estaba comentado por que toma mucho tiempo y se bloquea o no produce ningú
n resultado cuando se utilizan
# todos los datos de entrenamiento y todas las variables
# Para mostrar como funciona lo hacemos con los 5 predictores mas importantes
datatograph = ataques_10perc[['srv_count', 'count', 'src_bytes', 'dst_host_same_s
rc_port_rate', 'service_ecr_i']]
pd.plotting.scatter_matrix(datatograph, alpha = 0.3, figsize = (14,14), diagonal
= 'kde')
```

```
Out[77]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1a1eb96ef0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1b6c2160>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a121cac88>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1ea9f400>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1ea54048>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1a1ea54550>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1f07b668>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1eab2a58>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1eb83160>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1b614e48>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1a1ebdfa20>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1b685320>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a4eb11518>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a4eaf33c8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a4eaadc50>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1a1f0e37b8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a1f11def0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a2075f0b8>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a207adcc0>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a20f10978>],
                [<matplotlib.axes._subplots.AxesSubplot object at 0x1a20f50668>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a2123aa20>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a21298390>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a21309438>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x1a22ca0550>]], dtype=
          object)
```



Códificar los datos

Como se observó en las tablas anteriores, existen 4 variables categóricas entre los 42 predictores disponibles.

Para poder trabajar con los algoritmos, la mayoría por no decir todos deben tener los datos en formato númerico, de esta forma es más facil hacer la computación de lo que se busca.

Dicho lo anterior, creamos variables dummies para los atributos categóricos

In [29]:
```
# identificamos los posibles valores de cada predictor categórico
ataques_10perc.protocol_type.unique() # type of the protocol
#ataques_correg_test_10perc.protocol_type.unique()
```

Out[29]:
```
array(['tcp', 'udp', 'icmp'], dtype=object)
```

In [30]:
```
ataques_10perc.service.unique() # network service on the destination
```

Out[30]:
```
array(['http', 'smtp', 'finger', 'domain_u', 'auth', 'telnet', 'ftp',
       'eco_i', 'ntp_u', 'ecr_i', 'other', 'private', 'pop_3', 'ftp_data',
       'rje', 'time', 'mtp', 'link', 'remote_job', 'gopher', 'ssh', 'name',
       'whois', 'domain', 'login', 'imap4', 'daytime', 'ctf', 'nntp',
       'shell', 'IRC', 'nnsp', 'http_443', 'exec', 'printer', 'efs',
       'courier', 'uucp', 'klogin', 'kshell', 'echo', 'discard', 'systat',
       'supdup', 'iso_tsap', 'hostnames', 'csnet_ns', 'pop_2', 'sunrpc',
       'uucp_path', 'netbios_ns', 'netbios_ssn', 'netbios_dgm', 'sql_net',
       'vmnet', 'bgp', 'Z39_50', 'ldap', 'netstat', 'urh_i', 'X11',
       'urp_i', 'pm_dump', 'tftp_u', 'tim_i', 'red_i'], dtype=object)
```

In [31]:
```
len(ataques_correg_test_10perc.service.unique())
```

Out[31]:
```
65
```

In [32]:
```
ataques_10perc.flag.unique()
# ataques_correg_test_10perc.flag.unique()
```

Out[32]:
```
array(['SF', 'S1', 'REJ', 'S2', 'S0', 'S3', 'RSTO', 'RSTR', 'RSTOS0',
       'OTH', 'SH'], dtype=object)
```

In [33]:
```
EDA.describe_categorical(ataques_10perc)
```

	protocol_type	service	flag	attack_types	attack_category
count	145586	145586	145586	145586	145586
unique	3	66	11	23	5
top	tcp	http	SF	normal	normal
freq	130913	62054	87459	87832	87832

In [34]:
```
EDA.describe_categorical(ataques_correg_test_10percAll)
```

	protocol_type	service	flag	attack_types	attack_category
count	311029	311029	311029	311029	311029
unique	3	65	11	38	6
top	icmp	ecr_i	SF	smurf	dos
freq	164969	164352	248379	164091	223298

In [35]:
```
EDA.describe_categorical(ataques_correg_test_10perc)
```

	protocol_type	service	flag	attack_types	attack_category
count	77291	77291	77291	77291	77291
unique	3	65	11	38	6
top	tcp	http	SF	normal	normal
freq	71124	40350	53624	47913	47913

In [36]:
```
EDA.display_info_numericas(ataques_10perc)
```

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent
count	145586.000000	1.455860e+05	1.455860e+05	145586.000000	145586.000000	145586.000000
mean	132.025181	7.995700e+03	2.859780e+03	0.000137	0.020201	0.000048
std	1224.157053	1.820383e+06	6.080979e+04	0.011720	0.239368	0.010150
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000
25%	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000
50%	0.000000	1.470000e+02	1.050000e+02	0.000000	0.000000	0.000000
75%	0.000000	2.880000e+02	1.164750e+03	0.000000	0.000000	0.000000
max	58329.000000	6.933756e+08	5.155468e+06	1.000000	3.000000	3.000000

In [37]:
```
EDA.display_info_numericas(ataques_correg_test_10perc)
```

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	
count	77291.000000	7.729100e+04	7.729100e+04	77291.000000	77291.000000	77291.000000	772
mean	70.761512	3.427815e+03	2.870840e+03	0.000116	0.002536	0.000207	0.0
std	815.287350	2.554528e+05	3.222983e+04	0.010790	0.077607	0.019700	0.5
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.0
25%	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.0
50%	0.000000	2.110000e+02	3.210000e+02	0.000000	0.000000	0.000000	0.0
75%	0.000000	2.960000e+02	1.774000e+03	0.000000	0.000000	0.000000	0.0
max	57715.000000	6.282565e+07	5.203179e+06	1.000000	3.000000	3.000000	10

In [39]:
```
# Antes de crear las variables dummy, crearemos un dataframe con los dos conjunto
s de datos, tanto de entrenamiento
# como de validación; esto con el fin de utilizarlo mas adelante en la selección
de los mejores predictores
# y otros procesos

# X = ataques_10perc.copy()
# X.append(ataques_correg_test_10perc.copy(), ignore_index=True)

X = pd.concat([ataques_10perc, ataques_correg_test_10perc], axis=0, copy=True) #
Unimos las observaciones de ambos conjuntos
X.shape
```

Out[39]:
```
(222877, 44)
```

In [40]:
```
# Como todas las variables tienen más de 2 valores en sus atributos, podemos usar la función get_dummies de pandas
# De lo contrario solo debemos utilizar "factorize", para crear variables nominales binarias a numericas
# De igual modo para variables categóricas con más de 2 valores pero "ordinales"
categorical_variables = ['protocol_type', 'service', 'flag']
ataques_10perc = EDA.create_dummies(ataques_10perc, categorical_variables)
ataques_correg_test_10perc = EDA.create_dummies(ataques_correg_test_10perc, categorical_variables)
ataques_correg_test_10percAll = EDA.create_dummies(ataques_correg_test_10percAll, categorical_variables)
X = EDA.create_dummies(X, categorical_variables)
```

In [41]:
```
ataques_10perc.describe()
```

Out[41]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent
count	145586.000000	1.455860e+05	1.455860e+05	145586.000000	145586.000000	145586.000000
mean	132.025181	7.995700e+03	2.859780e+03	0.000137	0.020201	0.000048
std	1224.157053	1.820383e+06	6.080979e+04	0.011720	0.239368	0.010150
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000
25%	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000
50%	0.000000	1.470000e+02	1.050000e+02	0.000000	0.000000	0.000000
75%	0.000000	2.880000e+02	1.164750e+03	0.000000	0.000000	0.000000
max	58329.000000	6.933756e+08	5.155468e+06	1.000000	3.000000	3.000000

8 rows × 116 columns

In [42]:
```
ataques_correg_test_10perc.describe()
```

Out[42]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	
count	77291.000000	7.729100e+04	7.729100e+04	77291.000000	77291.000000	77291.000000	772
mean	70.761512	3.427815e+03	2.870840e+03	0.000116	0.002536	0.000207	0.0
std	815.287350	2.554528e+05	3.222983e+04	0.010790	0.077607	0.019700	0.5
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.0
25%	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000	0.0
50%	0.000000	2.110000e+02	3.210000e+02	0.000000	0.000000	0.000000	0.0
75%	0.000000	2.960000e+02	1.774000e+03	0.000000	0.000000	0.000000	0.0
max	57715.000000	6.282565e+07	5.203179e+06	1.000000	3.000000	3.000000	10

8 rows × 115 columns

Seleccionar los parámetros más importantes

Para la selección de los mejores atributos o predictores exiten varias técnicas, aquí solo aplicamos 2 que son eficientes computacionalmente. En el notebook o notas pasadas (Deteccion%20intrusos%20redes.ipynb#Selecci%C3%B3n-de-los-mejores-atributos-o-predictores) se llevaron a cabo algunos otros metodos.

Los resultados fueron distintos en todos los casos, lo que no es algo en lo que se pueda confiar y menos desconociendo la teoría o lógica del problema de redes de comunicación digital.

In [43]:
```
# Una forma rápida es gráficar o visualizar la matríz de correlaciones entre pred
ictores
corr = ataques_10perc.corr()
corr.head(40)
```

Out[43]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent
duration	1.000000	0.004280	0.002582	-0.001264	-0.009102	0.003707
src_bytes	0.004280	1.000000	-0.000162	-0.000051	-0.000365	-0.000018
dst_bytes	0.002582	-0.000162	1.000000	-0.000551	-0.003969	0.016147
land	-0.001264	-0.000051	-0.000551	1.000000	-0.000989	-0.000056
wrong_fragment	-0.009102	-0.000365	-0.003969	-0.000989	1.000000	-0.000400
urgent	0.003707	-0.000018	0.016147	-0.000056	-0.000400	1.000000
hot	0.009855	0.003999	0.000658	-0.000823	-0.005925	0.000142
num_failed_logins	0.004343	-0.000077	0.048789	-0.000211	-0.001521	0.141954
logged_in	-0.090667	-0.000928	0.028106	-0.011524	-0.082969	0.004818
num_compromised	0.061454	0.000025	0.022755	-0.000093	-0.000667	0.014268
root_shell	0.021395	-0.000075	0.031076	-0.000228	-0.001641	0.034730
su_attempted	0.058895	-0.000033	0.075442	-0.000101	-0.000727	-0.000041
num_root	0.059745	-0.000038	0.020425	-0.000122	-0.000877	0.009442
num_file_creations	0.078000	-0.000043	0.004285	-0.000243	-0.001746	0.015145
num_shells	-0.001435	-0.000045	-0.000454	-0.000214	-0.001542	-0.000087
num_access_files	0.023886	-0.000191	0.007099	-0.000596	-0.004294	0.019932
num_outbound_cmds	NaN	NaN	NaN	NaN	NaN	NaN
is_host_login	NaN	NaN	NaN	NaN	NaN	NaN
is_guest_login	0.020268	-0.000270	-0.000959	-0.000806	-0.005803	-0.000326
count	-0.078036	-0.001727	-0.032532	-0.008112	-0.013634	-0.003465
srv_count	-0.040818	-0.001438	-0.009519	-0.004276	0.095968	-0.001851
serror_rate	-0.069311	0.001787	-0.029805	0.017473	-0.041159	-0.003048
srv_serror_rate	-0.069252	0.001096	-0.029837	0.018309	-0.054248	-0.003045
rerror_rate	0.004626	0.000315	-0.016049	-0.002840	-0.029163	-0.001661
srv_rerror_rate	0.004850	0.001367	-0.015749	-0.004115	-0.029628	-0.001663
same_srv_rate	0.059618	-0.002217	0.036113	0.006868	0.051500	0.003656
diff_srv_rate	0.051852	0.007307	-0.014592	0.001307	-0.021783	-0.001596
srv_diff_host_rate	-0.039164	-0.001144	-0.003977	0.034247	-0.025112	-0.001822
dst_host_count	0.062703	-0.000117	-0.034975	-0.021110	0.039512	-0.006155
dst_host_srv_count	-0.116824	-0.003937	0.013327	-0.012802	-0.051512	-0.005306
dst_host_same_srv_rate	-0.120372	-0.002144	0.031658	0.007972	-0.051003	-0.003356
dst_host_diff_srv_rate	0.429391	0.000138	-0.016820	-0.002662	0.085303	0.013441
dst_host_same_src_port_rate	0.353500	0.005619	0.028689	0.037464	0.064637	0.002639
dst_host_srv_diff_host_rate	-0.028847	0.000760	0.001728	0.106807	-0.009676	-0.001494
dst_host_serror_rate	-0.067714	-0.001742	-0.028956	0.015462	-0.049449	-0.003053
dst_host_srv_serror_rate	-0.067908	0.001111	-0.028726	0.008634	-0.054156	-0.003040

In [44]:
```
f, ax = plt.subplots(figsize=(16, 16))
sns.heatmap(corr, mask=np.zeros_like(corr, dtype=np.bool), cmap=sns.diverging_pal
ette(220, 10, as_cmap=True),
            square=False, ax=ax)
```

Out[44]: <matplotlib.axes._subplots.AxesSubplot at 0x1a16fee978>

Aplicamos métodos automáticos para la selección de las variables

In [46]:
```
# Eliminamos los datos infinitos, nulos o errados para evitar errores en los algo
rtimos de ML
# X = EDA.clean_dataset(X)
```

In [47]:
```
# Teniendo en cuenta que los datos contienen las variables independientes o a pre
dicir
# Procedemos a removerlas (attack_types, attack_category, attack_category_bin)
y_all = X.attack_types.copy() # las 24 o 38 tipos de ataques
y_4f = X.attack_category.copy() # Solo 4 categorias
y_bin = X.attack_category_bin.copy() # Solo 2 categorias binarias

X.drop(['attack_types', 'attack_category', 'attack_category_bin'], axis=1, inplac
e=True)
```

In [48]:
```
X.shape
```

Out[48]: (222877, 116)

In [49]:
```
y_all.shape
```

Out[49]: (222877,)

In [50]:
```
# Seperamos los datos en entrenamiento 70% y validación 30%
# X = scale(X) # Normalizar o escalar los datos
X_train, X_test, y_train, y_test = train_test_split(X, y_bin, test_size=0.3, rand
om_state=1)
```

In [51]:
```
# Selección Univariada - Univariate Selection
t0 = time()
# feature extraction
test = SelectKBest(score_func=chi2, k=10)
fit = test.fit(X_train, y_train)
# summarize scores
np.set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X_train)
# summarize selected features
print(features[0:11,:])
print("Numero de variables seleccionadas:", features.shape[1])
print("Tiempo total: %.2fs" % (time() - t0))
```

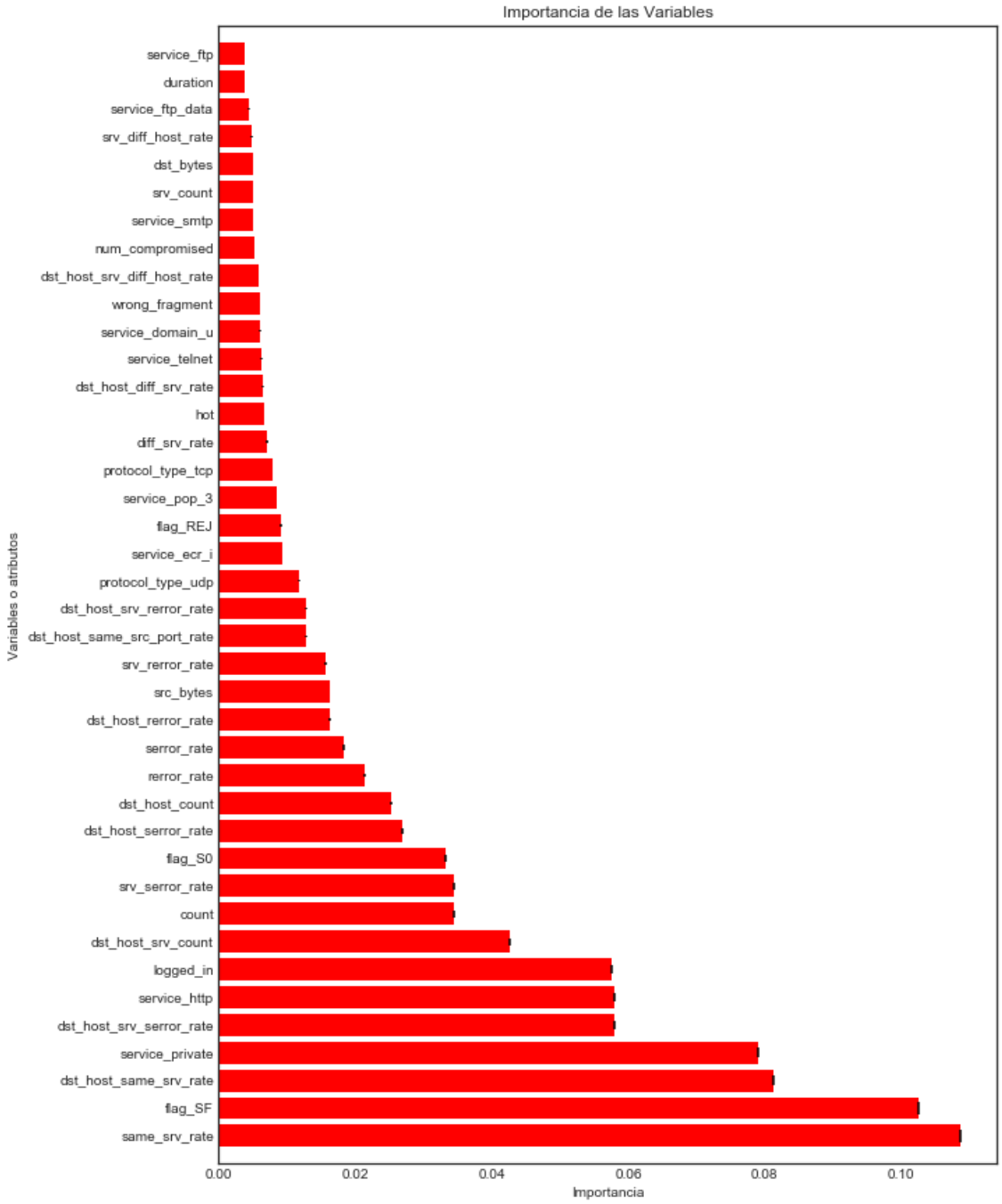
```
[  4.719e+05   1.372e+09   9.957e+07   3.112e+01   3.088e+03   2.179e+01
   4.444e+03   5.184e+02   4.214e+04   2.908e+02   1.738e+01   6.523e+00
   2.337e+03   9.588e+01   3.279e+00   2.463e+02         nan   4.924e+00
   2.688e+02   1.255e+07   7.880e+03   5.363e+04   5.361e+04   1.953e+04
   1.934e+04   3.373e+04   4.496e+03   5.747e+03   2.191e+06   9.011e+06
   3.502e+04   2.035e+03   1.070e+03   3.851e+02   5.327e+04   5.369e+04
   1.896e+04   1.911e+04   1.616e+02   3.650e+03   1.550e+00   1.603e+02
   4.824e-01   1.758e+02   1.898e+02   1.976e+02   1.665e+02   1.727e+02
   2.039e+02   1.841e+02   3.619e+03   2.101e+02   4.717e+02   1.385e+03
   1.805e+02   1.696e+02   7.346e+00   4.061e+02   4.886e+02   2.023e+02
   1.587e+02   4.044e+04   1.712e+02   1.285e+00   4.653e+02   1.867e+02
   1.634e+02   1.618e+02   1.805e+02   1.770e+02   1.665e+02   1.790e+02
   1.867e+02   1.572e+02   1.525e+02   1.805e+02   1.930e+02   1.696e+02
   1.712e+02   1.587e+02   1.973e+02   1.712e+01   1.603e+02   1.107e+03
   1.821e+02   6.576e+04         nan   2.081e+02   1.992e+02   1.852e+02
   4.960e+03   1.681e+02   1.754e+02   3.361e+02   1.649e+02   1.836e+02
   1.515e+03   6.426e-01   4.324e+00   7.587e+01   6.426e+00   2.120e+02
   1.758e+02   1.805e+02   1.696e+02   1.898e+02   1.657e+04   1.914e+03
   7.781e+00   1.012e+03   5.362e+04   1.089e+01   3.534e-02   2.732e+02
   4.171e+04   1.198e+02]
[[  0.000e+00   2.850e+02   1.345e+03   1.700e+01   0.000e+00   2.410e+02
    2.550e+02   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   1.920e+02   1.909e+03   9.000e+00   0.000e+00   2.100e+01
    2.550e+02   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   0.000e+00   0.000e+00   2.330e+02   1.000e+00   2.550e+02
    5.000e+00   1.000e+00   1.000e+00   1.000e+00]
 [  0.000e+00   3.230e+02   1.215e+03   1.500e+01   0.000e+00   1.300e+02
    2.550e+02   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   2.150e+02   8.750e+02   1.600e+01   0.000e+00   8.500e+01
    2.550e+02   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   0.000e+00   0.000e+00   2.240e+02   0.000e+00   2.550e+02
    6.000e+00   0.000e+00   1.000e+00   0.000e+00]
 [  0.000e+00   0.000e+00   0.000e+00   2.050e+02   1.000e+00   2.550e+02
    1.300e+01   1.000e+00   1.000e+00   1.000e+00]
 [  2.500e+01   1.880e+03   3.340e+02   1.000e+00   0.000e+00   9.400e+01
    2.010e+02   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   3.110e+02   1.745e+03   1.400e+01   0.000e+00   1.400e+01
    2.550e+02   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   1.460e+02   0.000e+00   1.000e+00   0.000e+00   2.550e+02
    1.000e+00   0.000e+00   0.000e+00   0.000e+00]
 [  0.000e+00   5.300e+01   5.000e+01   3.250e+02   0.000e+00   2.550e+02
    2.550e+02   0.000e+00   1.000e+00   0.000e+00]]
Numero de variables seleccionadas: 10
Tiempo total: 0.37s
```

In [52]:
```
# Utilizando Extra trees o random forest
t0 = time()
print(X_train.shape)

clf = ExtraTreesClassifier(n_jobs=-1)
clf = clf.fit(X_train, y_train)
print(clf.feature_importances_)

model = SelectFromModel(clf, prefit=True)
Xr_new = model.transform(X_train)
print("Cantidad de variables seleccionadas: ",Xr_new.shape[1])

print("Tiempo total: %.2fs" % (time() - t0))
```

```
(156013, 116)
[  2.924e-03   1.383e-02   3.904e-03   2.702e-05   7.564e-03   4.550e-05
   7.510e-03   8.888e-04   2.801e-02   6.325e-03   2.472e-04   2.315e-05
   1.695e-04   1.106e-04   1.397e-04   7.323e-05   0.000e+00   2.395e-05
   3.952e-03   9.300e-02   5.036e-03   6.009e-03   4.418e-02   4.267e-02
   2.099e-02   8.845e-02   1.352e-03   1.674e-03   1.366e-02   9.183e-03
   1.042e-01   7.241e-03   7.733e-03   7.280e-03   4.845e-02   1.056e-01
   6.131e-03   1.834e-02   1.466e-03   1.821e-02   6.481e-05   2.205e-07
   1.681e-04   0.000e+00   0.000e+00   2.246e-13   1.951e-12   0.000e+00
   4.606e-05   4.186e-05   5.043e-03   1.788e-09   3.060e-03   1.023e-02
   4.529e-10   0.000e+00   3.462e-04   9.775e-04   2.449e-03   8.388e-06
   1.971e-06   6.537e-02   1.256e-11   4.997e-06   4.261e-04   1.012e-08
   3.077e-10   0.000e+00   0.000e+00   2.710e-06   1.505e-05   3.597e-07
   5.327e-07   5.242e-07   0.000e+00   3.242e-07   2.557e-05   4.953e-08
   4.614e-06   1.081e-04   8.159e-04   4.203e-05   0.000e+00   8.706e-03
   2.253e-06   1.068e-02   0.000e+00   4.887e-06   6.129e-06   0.000e+00
   4.474e-03   1.304e-12   3.208e-06   6.966e-05   5.959e-07   2.289e-06
   7.479e-03   2.174e-06   3.916e-05   9.670e-05   3.590e-06   5.013e-04
   4.278e-06   0.000e+00   2.023e-06   1.977e-06   1.314e-02   1.932e-03
   1.653e-06   4.153e-03   1.415e-02   1.559e-04   5.048e-05   8.003e-05
   1.184e-01   1.503e-05]
Cantidad de variables seleccionadas:  21
Tiempo total: 1.72s
```

In [53]:
```
# Obtenemos las 40 variables mas importantes
num_var = 40
t0 = time()

# Build a forest and compute the feature importances
forest = ExtraTreesClassifier(n_estimators=250, random_state=0, n_jobs=-1)

forest.fit(X_train, y_train)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Ranking de Variables:")
for f in range(num_var): # X_train.shape[1]
    print("%d. %s - [variable %d] (%f)" % (f + 1, X_train.columns[indices[f]], in
dices[f], importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure(figsize=(10,15))
plt.title("Importancia de las Variables")
#plt.barh(range(X_train.shape[1]), importances[indices], color="r", yerr=std[indi
ces], align="center")
plt.barh(range(num_var), importances[indices[:num_var]], color="r", yerr=std[indi
ces[:num_var]], align="center")

#plt.xticks(range(X_train.shape[1]), indices)
#plt.yticks(range(X_train.shape[1]), X_train.columns[indices])
plt.yticks(range(num_var), X_train.columns[indices[:num_var]])
plt.xlabel("Importancia")
plt.ylabel("Variables o atributos")
#plt.ylim([-1, X_train.shape[1]])
plt.ylim([-1, num_var])
plt.show()

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Ranking de Variables:
1. same_srv_rate - [variable 25] (0.108615)
2. flag_SF - [variable 114] (0.102409)
3. dst_host_same_srv_rate - [variable 30] (0.081229)
4. service_private - [variable 85] (0.078984)
5. dst_host_srv_serror_rate - [variable 35] (0.058047)
6. service_http - [variable 61] (0.057966)
7. logged_in - [variable 8] (0.057477)
8. dst_host_srv_count - [variable 29] (0.042547)
9. count - [variable 19] (0.034548)
10. srv_serror_rate - [variable 22] (0.034384)
11. flag_S0 - [variable 110] (0.033258)
12. dst_host_serror_rate - [variable 34] (0.026892)
13. dst_host_count - [variable 28] (0.025181)
14. rerror_rate - [variable 23] (0.021388)
15. serror_rate - [variable 21] (0.018261)
16. dst_host_rerror_rate - [variable 36] (0.016254)
17. src_bytes - [variable 1] (0.016242)
18. srv_rerror_rate - [variable 24] (0.015658)
19. dst_host_same_src_port_rate - [variable 32] (0.012813)
20. dst_host_srv_rerror_rate - [variable 37] (0.012757)
21. protocol_type_udp - [variable 39] (0.011777)
22. service_ecr_i - [variable 53] (0.009342)
23. flag_REJ - [variable 106] (0.009160)
24. service_pop_3 - [variable 83] (0.008520)
25. protocol_type_tcp - [variable 38] (0.007778)
26. diff_srv_rate - [variable 26] (0.007066)
27. hot - [variable 6] (0.006721)
28. dst_host_diff_srv_rate - [variable 31] (0.006461)
29. service_telnet - [variable 96] (0.006288)
30. service_domain_u - [variable 50] (0.006117)
31. wrong_fragment - [variable 4] (0.006019)
32. dst_host_srv_diff_host_rate - [variable 33] (0.005766)
33. num_compromised - [variable 9] (0.005300)
34. service_smtp - [variable 90] (0.005078)
35. srv_count - [variable 20] (0.005001)
36. dst_bytes - [variable 2] (0.004939)
37. srv_diff_host_rate - [variable 27] (0.004740)
38. service_ftp_data - [variable 58] (0.004448)
39. duration - [variable 0] (0.003711)
40. service_ftp - [variable 57] (0.003702)
```




```
Tiempo total: 26.29s
```

In [54]:
```
X_train.columns[indices][:40]
```

Out[54]:
```
Index(['same_srv_rate', 'flag_SF', 'dst_host_same_srv_rate', 'service_private',
       'dst_host_srv_serror_rate', 'service_http', 'logged_in',
       'dst_host_srv_count', 'count', 'srv_serror_rate', 'flag_S0',
       'dst_host_serror_rate', 'dst_host_count', 'rerror_rate', 'serror_rate',
       'dst_host_rerror_rate', 'src_bytes', 'srv_rerror_rate',
       'dst_host_same_src_port_rate', 'dst_host_srv_rerror_rate',
       'protocol_type_udp', 'service_ecr_i', 'flag_REJ', 'service_pop_3',
       'protocol_type_tcp', 'diff_srv_rate', 'hot', 'dst_host_diff_srv_rate',
       'service_telnet', 'service_domain_u', 'wrong_fragment',
       'dst_host_srv_diff_host_rate', 'num_compromised', 'service_smtp',
       'srv_count', 'dst_bytes', 'srv_diff_host_rate', 'service_ftp_data',
       'duration', 'service_ftp'],
      dtype='object')
```

Separando el conjunto de datos de entrenamiento y de validación

Como hemos visto tenemos 4 conjuntos de datos de los cuales hemos ajustado para los diferentes procesos,

- **X** = Contiene el conjunto de datos de entrenamiento y validación juntos y depurados
- **y_all** = Contiene todos los datos de validación de la variable independiente, todas los tipos de ataques
- **y_4f** = Contiene los datos de validación de la variable independiente, las 4 categorias mas una adicional que nombre "unknown" o desconocida
- **y_bin** = Contiene los datos de validación de la variable independiente en forma binaria (0/1)
- **ataques_10perc** = Contiene los datos de entrenamiento sin duplicados y/o depurados
- **ataques_correg_test_10perc** = Contiene los datos de validación sin duplicados y/o depurados
- **ataques_correg_test_10percAll** = Contiene todos los datos de validación incluyendo duplicados

Ahora lo que haremos es seleccionar un conjunto de datos ajustados a las variables seleccionadas en el paso anterior con el fin de probar con algunos algoritmos. Se utilizarán todos los conjuntos de datos en mención para ver su precisión de incluir o no los datos duplicados y de incluir los datos de validación de la variable independiente de diferentes niveles, desde el más simple (buenas/malas conexiones) hasta identificar el tipo de ataque.

In [55]:
```
# Podriamos guardar estos conjuntos de datos para luego utilizarlos de manera más
rápida o dinámica
# EDA.save_data()
```

In [56]:
```
# Seleccion de los predictores
predictores_small = ['srv_count', 'count', 'src_bytes', 'dst_host_same_src_port_r
ate', 'service_ecr_i']
predictores15var = ['srv_count', 'service_ecr_i', 'dst_host_same_src_port_rate',
'count',
       'protocol_type_tcp', 'same_srv_rate', 'flag_SF',
       'dst_host_same_srv_rate', 'dst_host_srv_count', 'service_private',
       'logged_in', 'flag_S0', 'dst_host_srv_serror_rate', 'dst_host_count',
       'srv_serror_rate']

predictores40var = ['same_srv_rate', 'flag_SF', 'dst_host_same_srv_rate', 'servic
e_private',
       'dst_host_srv_serror_rate', 'service_http', 'logged_in',
       'dst_host_srv_count', 'count', 'srv_serror_rate', 'flag_S0',
       'dst_host_serror_rate', 'dst_host_count', 'rerror_rate', 'serror_rate',
       'dst_host_rerror_rate', 'src_bytes', 'srv_rerror_rate',
       'dst_host_same_src_port_rate', 'dst_host_srv_rerror_rate',
       'protocol_type_udp', 'service_ecr_i', 'flag_REJ', 'service_pop_3',
       'protocol_type_tcp', 'diff_srv_rate', 'hot', 'dst_host_diff_srv_rate',
       'service_telnet', 'service_domain_u', 'wrong_fragment',
       'dst_host_srv_diff_host_rate', 'num_compromised', 'service_smtp',
       'srv_count', 'dst_bytes', 'srv_diff_host_rate', 'service_ftp_data',
       'duration', 'service_ftp']

# ataques_10perc_train = pd.concat([ataques_10perc[ataques_10perc.columns[indices
][:40]], y_bin], axis=1)
```

In [57]:
```
# Definimos X (predictores) e y (respuesta)
# Training 1 - 5 variables y la independiente binaria
Xr5 = ataques_10perc[predictores_small]
yr5 = ataques_10perc.attack_category_bin

# Training 2 - 15 variables y la independiente binaria
Xr15 = ataques_10perc[predictores15var]
yr15 = ataques_10perc.attack_category_bin

# Training 3 - 40 variables y la independiente binaria
Xr40 = ataques_10perc[predictores40var]
yr40 = ataques_10perc.attack_category_bin

# Utilizamos todos los datos para poder comparar al final con la matríz de confus
ión del ganador del concurso KDDcup
# Test 1 - 5 variables y la independiente binaria
Xt5 = ataques_correg_test_10percAll[predictores_small]
yt5 = ataques_correg_test_10percAll.attack_category_bin

# Test 2 - 15 variables y la independiente binaria
Xt15 = ataques_correg_test_10percAll[predictores15var]
yt15 = ataques_correg_test_10percAll.attack_category_bin

# Test 3 - 40 variables y la independiente binaria
Xt40 = ataques_correg_test_10percAll[predictores40var]
yt40 = ataques_correg_test_10percAll.attack_category_bin
```

Selección de algoritmos y métodos

Aquí probaremos varios algoritmos y métodos para luego seleccionar el mejor o los mejores y realizar un emsanble o red neuronal.

Árbol de desición

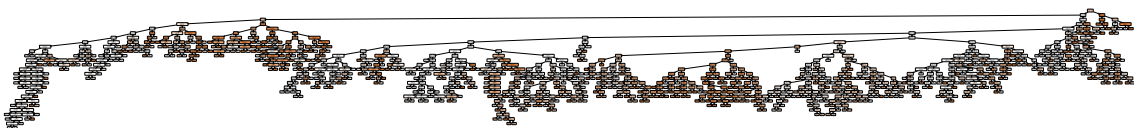
In [58]:
```
t0 = time()
# podriamos separar los datos en entrenamiento y validación, pero como ya los ten
emos en archivos o
# dataframe separados no es necesario realizar de nuevo este paso, sin temor a so
breajustar nuestro modelo
# X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=35)

# Para 5 variables importantes
#X_train = Xr5
#X_test = Xt5
#y_train = yr5
#y_test = yt5

# Árbol sin poda, como la variable independiente es númerica utilizamos un árbol
de regresión
treeclf = DecisionTreeRegressor(random_state=1) # max_depth=15 # DecisionTreeClas
sifier
treeclf.fit(Xr5,yr5)
# Verificamos la precisión de nuestro modelo, tanto como los datos de entrenamien
to y los de validación
print("Precisión Datos Entranamiento: ",treeclf.score(Xr5,yr5))
print("Precisión Datos Validación: ",treeclf.score(Xt5,yt5))

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión Datos Entranamiento:  0.990627988208
Precisión Datos Validación:  0.405702622389
Tiempo total: 0.28s
```

In [59]:
```
# Podemos gráficar el árbol si deseamos de la sgte forma:
t0 = time()

label_names = yt5.unique().tolist()

# Exportamos los datos del arbol a un archivo
export_graphviz(treeclf,out_file='./data/ataques_5var.dot',
                impurity=True, filled=True, rounded=True, special_characters=True
,
                feature_names=predictores_small,
                class_names= label_names )

# Leemos los datos guardados en el paso anterior
with open('./data/ataques_5var.dot') as f:
    dot_graph=f.read()

print("Tiempo total: %.2fs" % (time() - t0))
# Creamos la gráfica a partir de los datos anteriores
graphviz.Source(dot_graph)
```

```
Tiempo total: 0.28s
```

Out[59]:



Observamos que la precisión con un solo árbol y utilizando solo la variable independiente binaria no es muy buena

In [60]:
```
# Arbol de decisión con 15 variables
t0 = time()
# Ajustamos un árbol de clasificación con 15 niveles máximo (max_depth=15) sobre
todos los datos
treeclf = DecisionTreeClassifier(random_state=1)
treeclf.fit(Xr15,yr15)
# Verificamos la precisión de nuestro modelo, tanto como los datos de entrenamien
to y los de validación
print("Precisión Datos Entranamiento: ",treeclf.score(Xr15,yr15))
print("Precisión Datos Validación: ",treeclf.score(Xt15,yt15))

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión Datos Entranamiento:  0.99669611089
Precisión Datos Validación:  0.921541078163
Tiempo total: 0.53s
```

In [61]:
```
# Arbol de decisión con 40 variables
t0 = time()
# Ajustamos un árbol de clasificación con 15 niveles máximo (max_depth=15) sobre
todos los datos
treeclf = DecisionTreeClassifier(random_state=1)
treeclf.fit(Xr40,yr40)
# Verificamos la precisión de nuestro modelo, tanto como los datos de entrenamien
to y los de validación
print("Precisión Datos Entranamiento: ",treeclf.score(Xr40,yr40))
print("Precisión Datos Validación: ",treeclf.score(Xt40,yt40))

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión Datos Entranamiento:  0.999993131208
Precisión Datos Validación:  0.935716605204
Tiempo total: 1.27s
```

Observamos que la precisión aumenta un poco más a medida que incrementamos los predictores mas importantes, obtenidos en los pasos anteriores

In [62]:
```
# Hacemos prediciones sobre los datos de validación con el modelo de 40 variables
que dio mejor precisión
y40_pred_bin = treeclf.predict(Xt40)
# Calcular y desplegar la matríz de confusión
confusion_matrix(yt40, y40_pred_bin)
```

Out[62]:
```
array([[ 59643,    950],
       [ 19044, 231392]])
```

Comparamos nuestro primer modelo con los datos del ganador del concurso, mediante una validación cruzada Como se puede observar en el documento (http://cseweb.ucsd.edu/~elkan/clresults.html) de los resultados del ganador y del concurso, la matríz de confusión que se presenta es sobre los datos de la variable independiente tomando las 4 categorías de tipos de ataques.

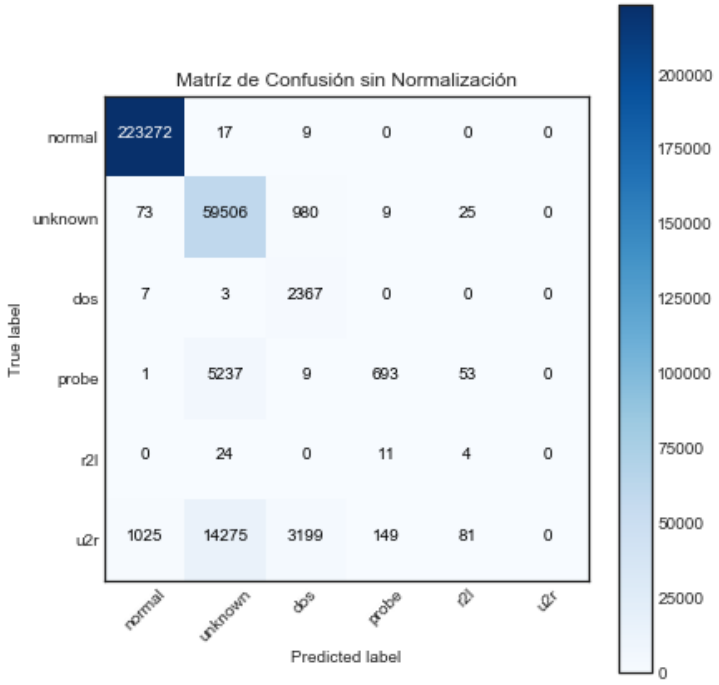
Procedemos entonces a calcularla con estos datos:

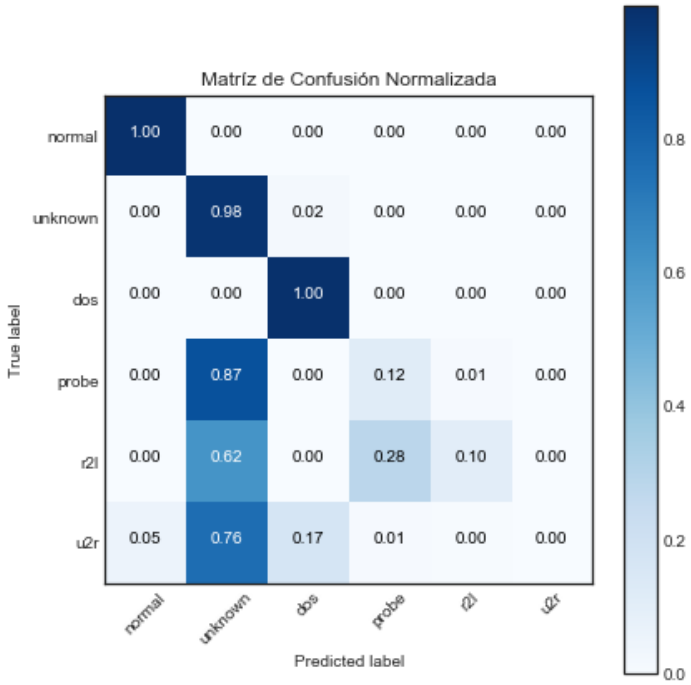
In [63]:
```
# Definimos X e y con las 40 variables y las independiente con 4 categorías
#Xr40 = ataques_10perc[predictores40var]
yr40 = ataques_10perc.attack_category
#Xt40 = ataques_correg_test_10percAll[predictores40var]
yt40 = ataques_correg_test_10percAll.attack_category

# Creamos el Árbol de decisión
t0 = time()
# Ajustamos un árbol de clasificación con 15 niveles máximo (max_depth=15) sobre
todos los datos
treeclf = DecisionTreeClassifier(random_state=1)
treeclf.fit(Xr40,yr40)
# Verificamos la precisión de nuestro modelo, tanto como los datos de entrenamien
to y los de validación
print("Precisión Datos Entranamiento: ",treeclf.score(Xr40,yr40))
print("Precisión Datos Validación: ",treeclf.score(Xt40,yt40))
print("-"*80)
# Hacemos prediciones sobre los datos de validación
y40_pred_4ctg = treeclf.predict(Xt40)
# Calcular y desplegar la matríz de confusión
cnf_matrix = confusion_matrix(yt40, y40_pred_4ctg)
print(cnf_matrix)
print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión Datos Entranamiento:  0.999993131208
Precisión Datos Validación:  0.919020412888
--------------------------------------------------------------------------------
[[223272     17      9      0      0      0]
 [    73  59506    980      9     25      0]
 [     7      3   2367      0      0      0]
 [     1   5237      9    693     53      0]
 [     0     24      0     11      4      0]
 [  1025  14275   3199    149     81      0]]
Tiempo total: 3.50s
```

In [64]:
```
np.set_printoptions(precision=2)

class_names = yt40.unique()

# Plot non-normalized confusion matrix
plt.figure(figsize=(6,6))
EDA.plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Matríz de Confusión sin Normalización')

# Plot normalized confusion matrix
plt.figure(figsize=(6,6))
EDA.plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,
                      title='Matríz de Confusión Normalizada')

plt.show()
```

```
Matríz de Confusión sin Normalización
[[223272     17      9      0      0      0]
 [    73  59506    980      9     25      0]
 [     7      3   2367      0      0      0]
 [     1   5237      9    693     53      0]
 [     0     24      0     11      4      0]
 [  1025  14275   3199    149     81      0]]
Matríz de Confusión Normalizada
[[  1.00e+00   7.61e-05   4.03e-05   0.00e+00   0.00e+00   0.00e+00]
 [  1.20e-03   9.82e-01   1.62e-02   1.49e-04   4.13e-04   0.00e+00]
 [  2.94e-03   1.26e-03   9.96e-01   0.00e+00   0.00e+00   0.00e+00]
 [  1.67e-04   8.74e-01   1.50e-03   1.16e-01   8.84e-03   0.00e+00]
 [  0.00e+00   6.15e-01   0.00e+00   2.82e-01   1.03e-01   0.00e+00]
 [  5.47e-02   7.62e-01   1.71e-01   7.96e-03   4.32e-03   0.00e+00]]
```





In [65]:
```
# Definimos X e y con las 40 variables y las independiente con todos los tipos de
ataque
#Xr40 = ataques_10perc[predictores40var]
yr40 = ataques_10perc.attack_types
#Xt40 = ataques_correg_test_10percAll[predictores40var]
yt40 = ataques_correg_test_10percAll.attack_types

# Creamos el Árbol de decisión
t0 = time()
# Ajustamos un árbol de clasificación con 15 niveles máximo (max_depth=15) sobre
todos los datos
treeclf = DecisionTreeClassifier(random_state=1)
treeclf.fit(Xr40,yr40)
# Verificamos la precisión de nuestro modelo, tanto como los datos de entrenamien
to y los de validación
print("Precisión Datos Entranamiento: ",treeclf.score(Xr40,yr40))
print("Precisión Datos Validación: ",treeclf.score(Xt40,yt40))
print("-"*80)
# Hacemos prediciones sobre los datos de validación
y40_pred_38ta = treeclf.predict(Xt40)
# Calcular y desplegar la matríz de confusión
cnf_matrix = confusion_matrix(yt40, y40_pred_38ta)
print(cnf_matrix)
print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión Datos Entranamiento:  0.999986262415
Precisión Datos Validación:  0.915519131657
--------------------------------------------------------------------------------
[[    0   467     0 ...,     0     0     0]
 [    0  1098     0 ...,     0     0     0]
 [    0     0     0 ...,     0     0     0]
 ...,
 [    0     0     0 ...,     0     0     0]
 [    0     0     0 ...,     0     0     0]
 [    0     0     1 ...,     0     0     0]]
Tiempo total: 4.66s
```

Observamos que la precisión con todos los tipos de ataques baja un poco para este primer modelo de Árboles de decisión.

Random Forest

In [79]:
```
# Al parecer esto toma mucho tiempo encontrar los mejores parametros.
# Como ya se habia ejecutado en otro cuaderno de notas, tomaremos los parametros
de ahí
t0 = time()

# Definimos X e y con las 40 variables y las independiente con 4 categorías
#Xr40 = ataques_10perc[predictores40var]
yr40 = ataques_10perc.attack_category
#Xt40 = ataques_correg_test_10percAll[predictores40var]
yt40 = ataques_correg_test_10percAll.attack_category

rfclf = RandomForestClassifier(n_jobs=-1) # utilizamos todos los procesadores y e
l GPU

# Escogemos algunas combinaciónes de parametros para escoger la mejor al final
parameters = {'n_estimators': [5, 10],
              'max_features': ['log2', 'sqrt','auto'],
              'criterion': ['entropy', 'gini'],
              'max_depth': [5, 10, 15],
              'min_samples_split': [3, 5],
              'min_samples_leaf': [1,5,8]
             }

# Tipo de puntaje usado para comparar la mejor combinación de parametros
acc_scorer = make_scorer(accuracy_score)

# Ejecutamos el metodo "grid search" o de busqueda de los mejores parametros para
el clasificador escogido
grid_obj = GridSearchCV(rfclf, parameters, scoring=acc_scorer)
grid_obj = grid_obj.fit(Xr40, yr40)

# Ajustamos el clasificador con la mejor combinación de parametros
rfclf = grid_obj.best_estimator_

# Ajustamos el mejor algortimo para los datos de las 40 variables
rfclf.fit(Xr40, yr40)
print(rfclf)

predictions = rfclf.predict(Xt40)
print("Precisión usando Random Forest: ", accuracy_score(yt40, predictions))

# Calcular y desplegar la matríz de confusión
cnf_matrix = confusion_matrix(yt40, predictions)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=5, max_features='sqrt', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=5, min_samples_split=5,
            min_weight_fraction_leaf=0.0, n_estimators=5, n_jobs=-1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
Precisión usando Random Forest:  0.388233894589
[[ 58502 164788      8      0      0      0]
 [    70  60360    163      0      0      0]
 [    10    477   1890      0      0      0]
 [     0   5993      0      0      0      0]
 [     0     39      0      0      0      0]
 [   217  17670    842      0      0      0]]
Tiempo total: 562.83s
```

In [67]:
```
t0 = time()
# Definimos X e y con las 40 variables y las independiente con 4 categorías
# Xr40 = ataques_10perc[predictores40var]
yr40 = ataques_10perc.attack_category
# Xt40 = ataques_correg_test_10percAll[predictores40var]
yt40 = ataques_correg_test_10percAll.attack_category


rfclf = RandomForestClassifier(bootstrap=False, class_weight=None, criterion='ent
ropy',
            max_depth=15, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=3,
            min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=-1,
            oob_score=False, random_state=None, verbose=0, warm_start=False)

# # Ajustamos el mejor algortimo para los datos de las 40 variables
rfclf.fit(Xr40, yr40)
predictions = rfclf.predict(Xt40)
print("Precisión usando Random Forest: ", accuracy_score(yt40, predictions))

# # Calcular y desplegar la matríz de confusión
cnf_matrix = confusion_matrix(yt40, predictions)
print(cnf_matrix)
print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión usando Random Forest:  0.921563584103
[[223274     10     14      0      0      0]
 [    70  60278    238      4      3      0]
 [     4      1   2372      0      0      0]
 [     0   5282      2    707      2      0]
 [     0     32      0      5      2      0]
 [   893  16888    940      4      4      0]]
Tiempo total: 3.10s
```

Random Forest con Boostrap

In [68]:
```
t0 = time()
# Definimos X e y con las 40 variables y las independiente con 4 categorías
# Xr40 = ataques_10perc[predictores40var]
yr40 = ataques_10perc.attack_category
# Xt40 = ataques_correg_test_10percAll[predictores40var]
yt40 = ataques_correg_test_10percAll.attack_category

# Modificamos la variables boostrap, criterion y el número de estimadores
rfclf = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini
',
            max_depth=15, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=3,
            min_weight_fraction_leaf=0.0, n_estimators=50, n_jobs=-1,
            oob_score=False, random_state=None, verbose=0, warm_start=False)

# Ajustamos el mejor algortimo para los datos de las 40 variables
rfclf.fit(Xr40, yr40)
predictions = rfclf.predict(Xt40)
print("Precisión usando Random Forest: ", accuracy_score(yt40, predictions))

# Calcular y desplegar la matríz de confusión
cnf_matrix = confusion_matrix(yt40, predictions)
print(cnf_matrix)
print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión usando Random Forest:  0.922303064988
[[223266     16     16      0      0      0]
 [    69  60298    223      2      1      0]
 [     3      6   2368      0      0      0]
 [     0   5064      0    928      1      0]
 [     0     34      0      2      3      0]
 [   540  17291    892      3      3      0]]
Tiempo total: 6.25s
```

Extra Trees Classification

In [69]:
```
# Extra Trees Classification
t0 = time()
seed = 7
num_trees = 100
max_features = 7
kfold = KFold(n_splits=10, random_state=seed)
model = ExtraTreesClassifier(n_estimators=num_trees, max_features=max_features)
results = cross_val_score(model, Xr40, yr40, cv=kfold)
print(results.mean())
model.fit(Xr40, yr40)

precision = model.score(Xt40, yt40)
print("Puntaje de Validación Extra Trees: ",precision)
# Predicciones
y_pred_class = model.predict(Xt40)
# Matríz de correlación
cnf_matrix = confusion_matrix(yt40, y_pred_class)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

```
0.991840055501
Puntaje de Validación Extra Trees:  0.920402920628
[[222772      7    519      0      0      0]
 [    72  60252    261      4      4      0]
 [     3     10   2364      0      0      0]
 [     0   5104      2    880      7      0]
 [     0     30      0      5      4      0]
 [   647  17028   1038      3     13      0]]
Tiempo total: 82.72s
```

Validación Cruzada con KFold

In [70]:
```
t0 = time()
run_kfold(rfclf, Xr40, yr40)
print("Tiempo total: %.2fs" % (time() - t0))
```

```
Grupo 1 precisión: 0.9998626279277423
Grupo 2 precisión: 0.9943677450374339
Grupo 3 precisión: 0.9937495707122742
Grupo 4 precisión: 0.9987636513496806
Grupo 5 precisión: 0.9995878837832268
Grupo 6 precisión: 0.9587196922865582
Grupo 7 precisión: 0.9980766588817145
Grupo 8 precisión: 0.997870586619041
Grupo 9 precisión: 0.9990383294408572
Grupo 10 precisión: 0.9992444017035307
Precisión promedio: 0.9939281147742058
Tiempo total: 42.63s
```

Árboles de Decisión con Bagging

In [71]:
```
t0 = time()

clf_bagging = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estima
tors=100, bootstrap=True,
                        random_state=42, n_jobs=-1, oob_score=True)

clf_bagging.fit(Xr40, yr40)
y_pred = clf_bagging.predict(Xt40)
#metrics.f1_score(y_pred, y_test)
print("Precisión usando Árboles con Bagging: ",metrics.accuracy_score(y_pred, yt4
0))

# Calcular y desplegar la matríz de confusión
cnf_matrix = confusion_matrix(yt40, y_pred)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Precisión usando Árboles con Bagging:  0.91930012957
[[223275     16      7      0      0      0]
 [    73  59533    978      5      4      0]
 [     8      1   2368      0      0      0]
 [     0   5235      9    746      3      0]
 [     0     28      0      4      7      0]
 [   825  14463   3427      4     10      0]]
Tiempo total: 54.68s
```

KNN con Bagging

In [78]:
```
# Bagging with KNN - toma algo de tiempo
t0 = time()

knnclf_bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5, max_f
eatures=0.5, n_estimators=10,
                                   bootstrap=True, random_state=42, n_jobs=-1, oob_s
core=False )

knnclf_bagging.fit(Xr40, yr40)
y_pred = knnclf_bagging.predict(Xt40)
print("Score Validación KNN con Bagging: ", metrics.accuracy_score(y_pred, yt40))

# Hacer las predicciones
y_pred_class = knnclf_bagging.predict(Xt40)
# Matríz de correlación
cnf_matrix = confusion_matrix(yt40, y_pred_class)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Score Validación KNN con Bagging:  0.918798568622
[[223240     53      5      0      0      0]
 [   149  60336    107      1      0      0]
 [   275     16   2086      0      0      0]
 [     0   5872     10    111      0      0]
 [     0     39      0      0      0      0]
 [   176  17851    702      0      0      0]]
Tiempo total: 989.03s
```

Gradient Boosting

In [72]:
```
# GBRT
gbclf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0, max_depth
=5, random_state=0)
gbclf.fit(Xr40, yr40)

t0 = time()
precision = gbclf.score(Xt40, yt40)
print("Score Validación GBRT: ",precision)
# Predicciones
y_pred_class = gbclf.predict(Xt40)
# Matríz de correlación
cnf_matrix = confusion_matrix(yt40, y_pred_class)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

```
Score Validación GBRT:  0.911136903633
[[222569     48    375    246     60      0]
 [   588  59349    108    474     74      0]
 [   362   1602    173     20    220      0]
 [     4   4558     94   1292     45      0]
 [     1     24      0      7      7      0]
 [   539  17859    137     66    128      0]]
Tiempo total: 8.30s
```

AdaBoost

In [73]:
```
# Utilizando AdaBoost
t0 = time()

#bdt_real = AdaBoostClassifier( DecisionTreeClassifier(max_depth=15), n_estimator
s=500, learning_rate=1)
#bdt_discrete = AdaBoostClassifier( DecisionTreeClassifier(max_depth=15), n_estim
ators=500, learning_rate=1.5,
#    algorithm="SAMME")

# Create and fit an AdaBoosted decision tree
bdt = AdaBoostClassifier(DecisionTreeClassifier(max_depth=15), n_estimators=100)
# algorithm="SAMME", algorithm="SAMME.R", learning_rate=1.0

bdt.fit(Xr40, yr40)
scores = cross_val_score(bdt, Xr40, yr40)
print(scores.mean())

print("Tiempo total: %.2fs" % (time() - t0))

y_pred = bdt.predict(Xt40)
print(metrics.accuracy_score(y_pred, yt40))

t0 = time()
precision = bdt.score(Xt40, yt40)
print("Score Validación AdaBoost: ",precision)

# Predicciones
y_pred_class = bdt.predict(Xt40)
# Matríz de correlación
cnf_matrix = confusion_matrix(yt40, y_pred_class)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

```
0.925780332751
Tiempo total: 230.82s
0.920968784261
Score Validación AdaBoost:  0.920968784261
[[223268      9     21      0      0      0]
 [    71  60292    225      2      3      0]
 [     3      1   2373      0      0      0]
 [     0   5478      0    511      4      0]
 [     0     33      0      2      4      0]
 [   732  17227    762      4      4      0]]
Tiempo total: 17.96s
```

Staking o Voting

In [74]:
```
# from sklearn.svm import SVC

# Training classifiers
t0 = time()
clf1 = DecisionTreeClassifier(max_depth=15)
clf2 = KNeighborsClassifier(n_neighbors=5)
#clf3 = SVC(kernel='rbf', probability=True)
#eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2), ('svc', clf3)],
voting='soft', weights=[2,1,2])
eclf = VotingClassifier(estimators=[('dt', clf1), ('knn', clf2)], voting='soft',
weights=[1,1])

clf1 = clf1.fit(Xr40, yr40)
clf2 = clf2.fit(Xr40, yr40)
#clf3 = clf3.fit(Xr40, yr40) # Esto toma demasiado tiempo

eclf = eclf.fit(Xr40, yr40)
print("Tiempo total: %.2fs" % (time() - t0))

t0 = time()

# Validamos los datos de prueba
precision = eclf.score(Xt40, yt40)
print("Score Validación Voting: ",precision)
# make predictions for testing set
y_pred_class = eclf.predict(Xt40)
# Matríz de correlación
cnf_matrix = confusion_matrix(yt40, y_pred_class)
print(cnf_matrix)

print("Tiempo total: %.2fs" % (time() - t0))
```

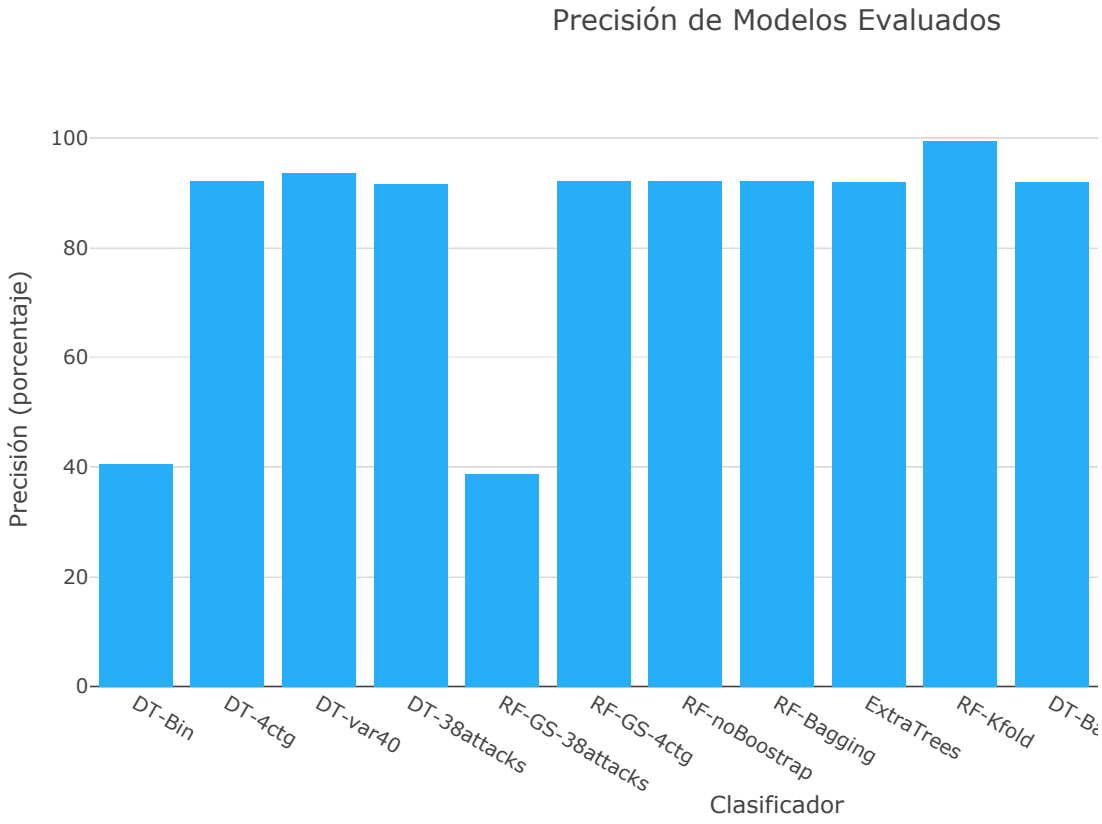
```
Tiempo total: 36.53s
Score Validación Voting:  0.920377199554
[[223265     19     14      0      0      0]
 [   165  60283    142      3      0      0]
 [   150      4   2223      0      0      0]
 [     0   5196    302    489      6      0]
 [     0     32      0      3      4      0]
 [  1027  16926    775      1      0      0]]
Tiempo total: 50.75s
```

Como se aprecia, no se utilizaron métodos rigurozamente escogidos, sino solo dos para mostrar la bonda del clasificador.
Se debe realizar un clasificador con los mejores vistos arriba y mirar que sucede con la presición del modelo.

Resumen de los métodos utilizados

MODELO	Validación
1. Árbol de desición (clasif. binario)	0.405702622389
2. Árbol de desición (4 categ.)	0.921541078163
3. Árbol de desición (40 var 4 ctg)	0.935716605204
4. Árbol de desición (40 var 38 tipos ataques)	0.915519131657
5. Random Forest (GridSearchCV, 38 attacks) *	0.388233894589
5. Random Forest (GridSearchCV, 4 ctg)	0.921586090043
6. Random Forest (Sin Bootstrap)	0.921563584103
7. Random Forest (Con Boostrap)	0.922303064988
8. Extra Trees	0.920402920628
9. Random Forest con KFold	0.993928114775
10. Árboles de Decisión con Bagging	0.919300129570
11. KNN con Bagging	0.918798568622
12. Gradient Boosting Regression Trees	0.911136903633
13. Árboles de Decisión con AdaBoost	0.920968784261
14. Voting (DT, KNN)	0.920377199554

In [86]:
```
#import plotly
#from plotly.graph_objs import Bar, Scatter, Layout, Figure
#from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
#plotly.offline.init_notebook_mode(connected=True)

Xdata = ['DT-Bin', 'DT-4ctg', 'DT-var40', 'DT-38attacks', 'RF-GS-38attacks', 'RF-
GS-4ctg', 'RF-noBoostrap',
         'RF-Bagging', 'ExtraTrees', 'RF-Kfold', 'DT-Bagging', 'KNN-Bagging', 'GB
RT', 'DT-AdaBoost', 'Voting-DT-KNN']
Ydata = [0.405702622389, 0.921541078163,0.935716605204,0.915519131657,0.388233894
589,0.921586090043,0.921563584103,
         0.922303064988,0.920402920628,0.993928114775,0.919300129570,0.9187985686
22,0.911136903633,0.920968784261,
         0.920377199554]
Ydata = np.dot(Ydata, 100)

clasificadores = Bar(x=Xdata, y=Ydata, name='Modelo', marker=dict(color='#26aff8
') )

data = [clasificadores]
layout = Layout(title="Precisión de Modelos Evaluados", xaxis=dict(title='Clasifi
cador'),
                yaxis=dict(title='Precisión (porcentaje)'))
fig = Figure(data=data, layout=layout)

iplot(fig)
```



Comparación de resultados con el ganador del KDDCup

Ya tenemos unos resultados que nos pueden orientar en el mejor clasificador a utilizar para comparar con el método del ganador del KDDCup del 99. Se debe tener en cuenta que solo se han utilizado con algunos parametros basicos y con el juego de datos del 10% que ofrecen.

Los métodos con mayor presición fueron **Árboles de Decisión con Bagging (93.57%)** y **Random Forest con KFold (%99.39)**; Aunque estos fueron los que obtuvieron mayores puntajes, es necesario evaluarlos de nuevo, por que el DT podría estar sobreajustado y el Kfold no se ha implementado sobre los datos de validación al parecer.

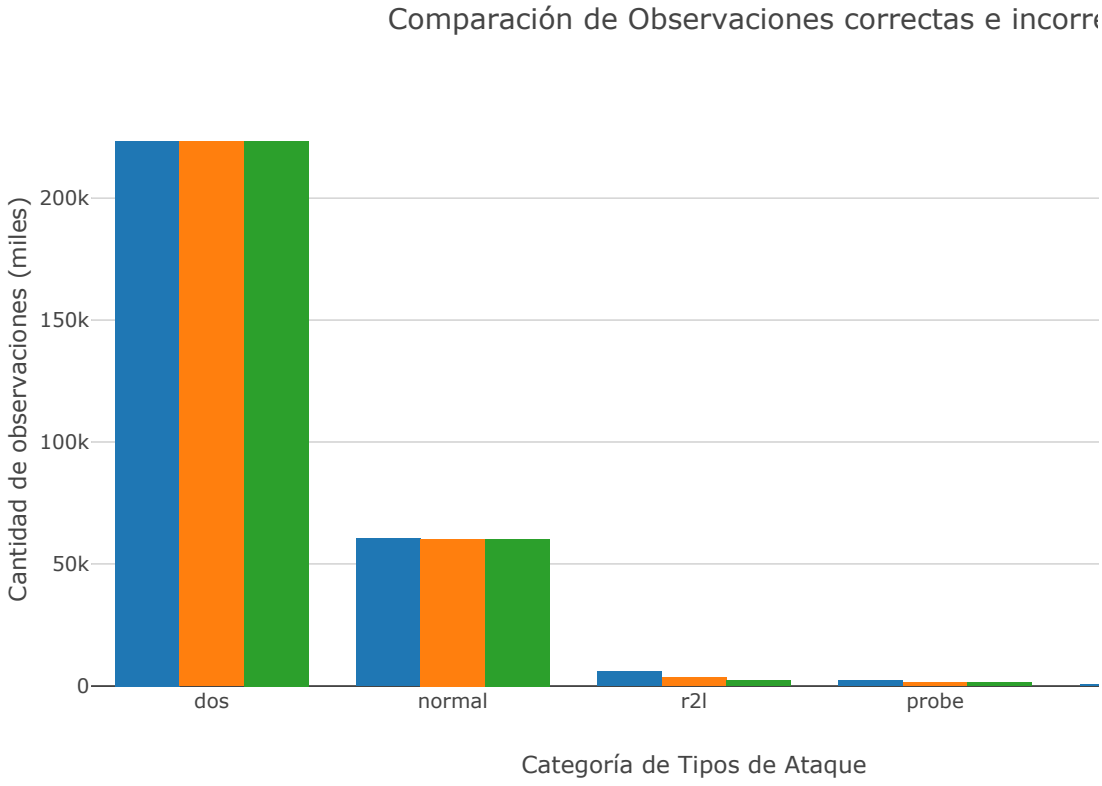
Por el momento, aquí mostramos los resultados de Random Forest con **Boosting and Bagging (92.23%)**:

predicted	dos	normal	probe	r2l	u2r	unknown	correct
dos	**223266**	16	16	0	0	0	99.99 %
normal	69	**60298**	223	2	1	0	99.51 %
probe	3	6	**2368**	0	0	0	99.62 %
r2l	0	5064	0	**928**	1	0	15.48 %
u2r	0	34	0	2	**3**	0	7.69 %
unknown	540	17291	892	3	3	**0**	0%

Es necesario hacer el ejercicio con todos los datos ofrecidos y mirar la precisión de nuevo, esto puede realizarse con 2 a 3 metodos de los anterior descritos.

In [151]:
```
#yt40.sort_values().unique()
#yt40.value_counts()
```

In [150]:
```
#labels = ["dos", "normal", "unknown", "r2l", "probe", "u2r"]
#reales = [223298, 60593, 18729, 5993, 2377, 39]
labels = ["dos", "normal", "r2l", "probe", "u2r"]
reales = [223298, 60593, 5993, 2377, 39]
winner = [223226, 60262, 3471, 1360, 30]
solucion1 = [223266, 60298, 2368, 928, 3]


trace1 = Bar( x=labels, y=reales, name='Reales' )
trace2 = Bar( x=labels, y=winner, name='Ganador KDDCup')
trace3 = Bar( x=labels, y=solucion1, name='Mi Solución')

data = [trace1, trace2, trace3]
layout = Layout( barmode='group')

layout = Layout(barmode='group', title="Comparación de Observaciones correctas e
incorrectas", xaxis=dict(title='Categoría de Tipos de Ataque'),
                yaxis=dict(title='Cantidad de observaciones (miles)'))
fig = Figure(data=data, layout=layout)

iplot(fig)
```



PERFORMANCE OF THE WINNING ENTRY

The winning entry achieved an average cost of 0.2331 per test example and obtained the following confusion matrix:

predicted	0	1	2	3	4	correct
0	60262	243	78	4	6	99.5%
1	511	3471	184	0	0	83.3%
2	5299	1328	223226	0	0	97.1%
3	168	20	0	30	10	13.2%
4	14527	294	0	8	1360	8.4%
---	---	---	---	---	---	---
correct	74.6%	64.8%	99.9%	71.4%	98.8%	---

In the table above the five attack categories are numbered as follows:

código	tipo de ataque
0	normal
1	probe
2	denial of service (DOS)
3	user-to-root (U2R)
4	remote-to-local (R2L)

We briefly describe our approach (https://www.researchgate.net/publication/220520055_Winning_the_KDD99_Classification_Cup_Bagged_Boosting) for the KDD99 Classification Cup. The solution is essentially a mixture of bagging and boosting. Additionally, asymmetric error costs are taken into account by minimizing the so-called conditional risk. Furthermore, the standard sampling with replacement methodology of bagging was modified to put a specific focus on the smaller but expensive-if-predicted-wrongly classes.