Detección de patrones maliciosos dentro de la red

Fase #2

Recordemos el objetivo…

Crear un modelo para que **detecte patrones** dentro de la red haciendo uso de la **actividad** (logs) registrados en la nube y así identificar posibles ataques. El propósito es **mantener segura la información de los usuarios** conectados a la red, **alertar** a los administradores en caso de encontrar una actividad sospechosa y **bloquear** al usuario detectado como malicioso.

¿Cómo se logrará el objetivo?

Se entrenarán tres modelos diferentes y se espera que al hacer uso de los tres modelos de forma seguida (el output de uno es el input del siguiente) se obtengan mejores resultados.

El resultado esperado es tener la cantidad mínima de falsos negativos y falsos positivos, aumentar la precisión de la clasificación y tener un resultado confiable.

Haciendo uso de dos datasets

Análisis exploratorio

Se hizo uso de las columnas con una correlación arriba de 0.5 y luego se seleccionaron las columnas categóricas que se consideraron de mayor relevancia para entrenar los modelos.

Variables categóricas usadas:

- Dataset 1: *protocol_type*, *service*
- Dataset 2: *PacketLengthClassification*, *IdleMeanClassification*

Final data - dataset 1

```
final_data_1 = pd.concat([str_df, filtered_data], axis=1)
print('Number of columns:', len(final_data_1.columns))
final_data_1.head()
```

```
Number of columns: 13
```

	protocol_type	service	flag	class	count	serror_rate	srv_serror_rate	same_srv_rate	dst_host_srv_count	dst_host_same_srv_rate	dst_host_serror_rate	dst_host_srv_serror_rate	isAttack
0	tcp	ftp_data	SF	normal	2.0	0.0	0.0	1.00	25.0	0.17	0.00	0.00	0
1	udp	other	SF	normal	13.0	0.0	0.0	0.08	1.0	0.00	0.00	0.00	0
2	tcp	private	S0	anomaly	123.0	1.0	1.0	0.05	26.0	0.10	1.00	1.00	1
3	tcp	http	SF	normal	5.0	0.2	0.2	1.00	255.0	1.00	0.03	0.01	0
4	tcp	http	SF	normal	30.0	0.0	0.0	1.00	255.0	1.00	0.00	0.00	0

Final data - dataset 2

```
print("columns: ", str(len(filtered_data.columns)))
filtered_data.head()
```

✓ 0.0s Python

```
columns:  18
```

	BwdPacketLengthMax	BwdPacketLengthMean	BwdPacketLengthStd	FlowIATStd	FlowIATMax	FwdIATTotal	FwdIATStd	FwdIATMax	MaxPacketLength	PacketLengthMean	PacketLengthStd	PacketI
0	179	89.5	103.345698	135.557286	445	640	194.325157	497	220	66.500000	99.001837	
1	1472	736.0	849.859596	192.795228	684	900	252.411229	734	1472	253.142857	527.434262	
2	1415	707.5	816.950631	236.433336	777	1205	397.058392	1008	1415	467.166667	690.098917	
3	185	92.5	106.809800	90.767652	299	511	131.900594	349	226	68.500000	101.933579	
4	1472	736.0	849.859596	148.698266	531	773	196.665733	580	1472	254.000000	527.520762	

Modelos utilizados y resultados: dataset 1

K-NN

Dataset 1 *pred*

```
TP:  19612
FP:  491
FN:  870
TN:  16504

 [[19612   491]
 [  870 16504]]

Accuracy: 0.9636843930944312
Precision: 0.9711091497499265
Recall:  0.9499251755496719
F1:  0.9604003607902469
```

Dataset 1 *val*

```
TP:  9946
FP:  221
FN:  404
TN:  8325

 [[9946  221]
 [ 404 8325]]
```

```
------------ On val -------------
Accuracy: 0.966924216765453
Precision: 0.9741399485139246
Recall:  0.953717493412762
F1:  0.963820549927641
```

Árbol de decisión

Dataset 1 *pred*

```
TP:  19601
FP:  364
FN:  1052
TN:  16460

 [[19601   364]
 [ 1052 16460]]
```

```
Accuracy: 0.962216826320143
Precision: 0.9783642415596766
Recall:  0.9399269072635906
F1:  0.9587604846225536
```

Dataset 1 *val*

```
------------ On val -------------
TP:  10009
FP:  158
FN:  481
TN:  8248

 [[10009   158]
 [  481  8248]]
```

```
------------ On val -------------
Accuracy: 0.9661833192209992
Precision: 0.9812039019747799
Recall:  0.9448963226028182
F1:  0.9627079077910708
```

SVM

Dataset 1 *pred*

```
TP:  19215
FP:  850
FN:  1000
TN:  16412

 [[19215   850]
 [ 1000 16412]]
```

```
Accuracy: 0.9506363903193958
Precision: 0.9507588923647318
Recall:  0.9425683436710315
F1:  0.9466459018284594
```

Dataset 1 *val*

```
------------ On val -------------
TP:  9794
FP:  373
FN:  478
TN:  8251

 [[9794  373]
 [ 478 8251]]

------------ On val -------------
Accuracy: 0.9549640135478408
Precision: 0.9567486085343229
Recall:  0.9452400045824264
F1:  0.9509594882729212
```

Modelo final:
Pipeline de modelos

Dataset 1: KNN -> Árbol de decisión -> SVM

Data de predicción

```
TP:  19686
FP:  379
FN:  935
TN:  16477

 [[19686   379]
 [  935 16477]]
```

```
Accuracy: 0.9649384956106412
Precision: 0.977515424774561
Recall:  0.9463014013324145
F1:  0.9616551885140656
```

Data de validación

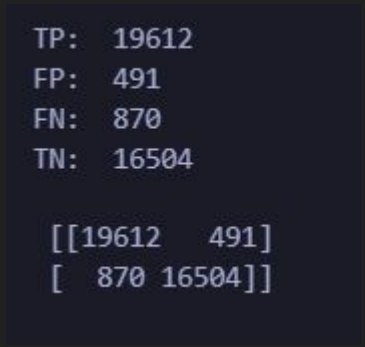
```
------------ On val -------------
TP:  9794
FP:  373
FN:  478
TN:  8251

 [[9794  373]
 [ 478 8251]]
```

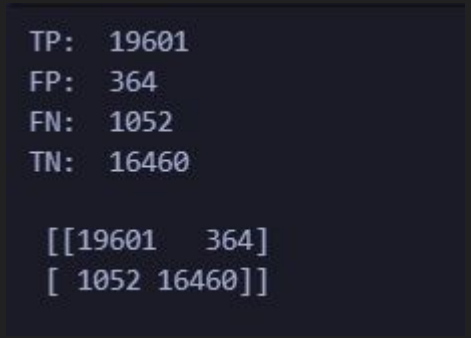
```
------------ On val -------------
Accuracy: 0.9549640135478408
Precision: 0.9567486085343229
Recall:  0.9452400045824264
F1:  0.9509594882729212
```

Dataset 1: KNN -> Árbol de decisión -> SVM

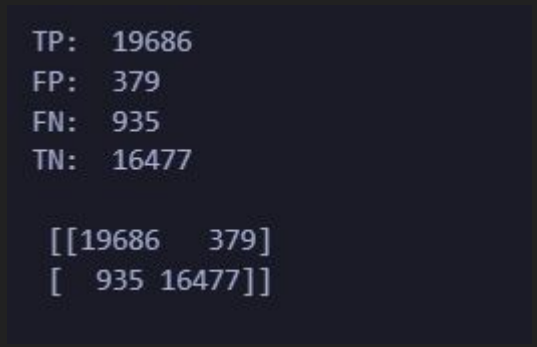
No se consiguió una mejora significativa al unir los tres modelos. Ya que el resultado es casi similar al de KNN o árbol de decisión por sí solos. Pero, hubo mejora en la clasificación de FP al compararlo con KNN y de FN al compararlo con árbol de decisión.



KNN



Árbol



Pipeline

Modelos utilizados y resultados: dataset 2

K-NN

Dataset 2 *pred*

```
TP:  14413
FP:  664
FN:  380
TN:  16043

array([[14413,   664],
       [  380, 16043]], dtype=int64)
```

```
Accuracy: 0.9664761904761905
Precision: 0.9582513270113914
Recall:  0.9783230834804847
F1:  0.9681831877071406
```

Dataset 2 *val*

```
------------ On val -------------
TP:  7297
FP:  352
FN:  190
TN:  8184

array([[7297,  352],
       [ 190, 8184]], dtype=int64)
```

```
------------ On val -------------
Accuracy: 0.9662984459838981
Precision: 0.9579144259995324
Recall:  0.9785048961069979
F1:  0.9681001890359169
```

Árbol de decisión

Dataset 2 *pred*

```
------------ On val -------------
TP:  6846
FP:  803
FN:  345
TN:  8029

array([[6846,  803],
       [ 345, 8029]], dtype=int64)
```

```
------------ On val -------------
Accuracy: 0.928352992573176
Precision: 0.9090806159420289
Recall:  0.9588010508717458
F1:  0.9332790886899918
```

Dataset 2 *val*

```
TP:  13643
FP:  1434
FN:  792
TN:  15631

array([[13643,  1434],
       [  792, 15631]], dtype=int64)
```

```
Accuracy: 0.9293333333333333
Precision: 0.9159683562847935
Recall:  0.9517749497655726
F1:  0.9335284280936454
```

Random Forest Classifier

Dataset 2 *pred*

```
TP:  13624
FP:  1453
FN:  632
TN:  15791

array([[13624,  1453],
       [  632, 15791]], dtype=int64)
Accuracy: 0.9338095238095238
Precision: 0.9157388077012294
Recall:  0.9615173841563661
F1:  0.938069920099801
```

Dataset 2 *val*

```
------------ On val -------------
TP:  6878
FP:  771
FN:  339
TN:  8035

array([[6878,  771],
       [ 339, 8035]], dtype=int64)

Accuracy: 0.9346564313798914
Precision: 0.9143762017871282
Recall:  0.9653689992834965
F1:  0.9391809468486784
```

Modelo final:
Pipeline de modelos

Dataset 2: KNN -> Árbol de decisión -> Random Forest

Data de predicción

```
TP:  14778
FP:  299
FN:  1483
TN:  14940

array([[14778,   299],
       [ 1483, 14940]], dtype=int64)
```

```
Accuracy: 0.9434285714285714
Precision: 0.9803792899796575
Recall:  0.9096998112403337
F1:  0.9437180216031837
```

Data de validación

```
TP:  7496
FP:  153
FN:  742
TN:  7632

array([[7496,  153],
       [ 742, 7632]], dtype=int64)
```

```
------------ On val -------------
Accuracy: 0.9441427947325719
Precision: 0.9803468208092485
Recall:  0.9113924050632911
F1:  0.9446129092146791
```

Dataset 2: KNN -> Árbol de decisión -> Random Forest

No se consiguió una mejora significativa al unir los tres modelos. Ya que el resultado es incluso menos preciso que el KNN por sí solo. Pero, al comparar la matriz de confusión se puede observar que se clasificaron de forma incorrecta más FN, lo cual es *mejor* que clasificar de forma incorrecta FP.

```
TP:  14413
FP:  664
FN:  380
TN:  16043

array([[14413,   664],
       [  380, 16043]], dtype=int64)
```

KNN

```
TP:  14778
FP:  299
FN:  1483
TN:  14940

array([[14778,   299],
       [ 1483, 14940]], dtype=int64)
```

Pipeline