Local Outlier Factor

El algoritmo Local Outlier Factor (LOF) es un método de detección de anomalías no supervisado que calcula la desviación de densidad local de un punto de datos dado con respecto a sus vecinos. Considera como valores atípicos las muestras que tienen una densidad sustancialmente más baja que sus vecinas.

```
#Importamos Librerias

# Tratamiento de datos
# ==============================================================================
import numpy as np
import pandas as pd
from mat4py import loadmat
from sklearn.datasets import make_blobs
from numpy import where

# Gráficos
# ==============================================================================
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sns
#style.use('ggplot') or plt.style.use('ggplot')

# Preprocesado y modelado
# ==============================================================================
from sklearn.ensemble import IsolationForest
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import OneClassSVM
from sklearn.metrics import confusion_matrix
# Configuración warnings
# ==============================================================================
import warnings
warnings.filterwarnings('ignore')


#importamos dataset para el training. Usaremos una muestra de poco más de 10mil observacic
data = pd.read_csv('muestra1.csv')
data.head(10)
```

	Unnamed: 0	Duration	Proto	SrcPt	DstPt	Packets	Bytes	Flags	Tos	class	Pac
0	1	0.000	6	33877	443.0	1	66.0	16	0	0	1(
1	2	0.042	6	80	52335.0	2	413.0	26	32	0	
2	3	0.215	6	51696	80.0	8	1350.0	26	0	0	
3	4	0.143	6	37251	443.0	4	1810.0	24	0	0	

```
# Seleccionamos las features que usaremos en el entrenamiento. He hecho pruebas cambiando
#La mejor combinación de features para este algoritmo es la siguiente
from numpy import quantile, where, random

# model specification
from sklearn.neighbors import LocalOutlierFactor
data = pd.read_csv('muestra1.csv')
datos_X1 = pd.DataFrame(data[['SrcPt', 'Flags', 'Tos', 'Proto', 'DstPt']])
datos_y1 = pd.DataFrame(data[['class']])

model = LocalOutlierFactor(n_neighbors=2)
model.fit_predict(X=datos_X1)
lof = model.negative_outlier_factor_
cuantil_01 = quantile(lof, .22)
score = pd.DataFrame(lof)
anomalia = pd.DataFrame(datos_y1)
df_resultados = score.join(anomalia)
df_resultados = df_resultados.rename(columns={0:'score', 'class':'anomalia'})
df_resultados = df_resultados \
                .sort_values('score', ascending=True) \
                .reset_index(drop=True)

df_resultados['clasificacion'] = np.where(df_resultados.score <= cuantil_01, 1, 0)
confusion_matrix(df_resultados.anomalia, df_resultados.clasificacion)
```

```
array([[7217, 2032],
       [ 962,  275]], dtype=int64)
```

Resultados no muy buenos, sobre todo para la predicción de anomalías que no llega ni al 50%

K-means

K-means es un algoritmo de clasificación no supervisada (clusterización) que agrupa objetos en k grupos basándose en sus características. El agrupamiento se realiza minimizando la suma de distancias entre cada objeto y el centroide de su grupo o cluster. Se suele usar la distancia cuadrática.

```
from sklearn.cluster import KMeans
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
```

```
from matplotlib import pyplot as plt
%matplotlib inline
from sklearn.metrics import confusion_matrix

#Cargamos el dataset y lo asignamos a la variable data
data = pd.read_csv('muestra1.csv')
#seleccionamos las variables que usaremos en el análisis. He hecho pruebas con diferentes
#genera mejores resultados predictivos para ambas clases
df = data[['Duration', 'Proto', 'SrcPt', 'DstPt']]
#Definimos el número de clusters, en este caso al ser 2 clases, necesitamos 2 clusters
km = KMeans(n_clusters=2)
#Hacemos un fit para obtener las predicciones
y_predicted = km.fit_predict(df)
#creamos un dataframe con las prediciones generadas
y_pred = pd.DataFrame(y_predicted)
#definimos la variable objetivo
y_train = data[["class"]]
#calculamos la matrix de confusión tomando como input las predicciones y los valores reale
confusion_matrix(y_train, y_pred)
```

```
array([[4616, 4633],
       [ 637,  600]], dtype=int64)
```

Pésimos resultados, al probar con otros sets de variables, cuando se mejora una clase, se empeora la predicción de la otra, al final, este fue lo mejor que pude obtener con K-means, quizás para un estudio multiclase (de 3 o más clusters) podría ser más interesante.

Isolation Forest

Al igual que en Random Forest, un modelo Isolation Forest está formado por la combinación de múltiples árboles llamados isolation trees. Estos árboles se crea de forma similar a los de clasificación-regresión: las observaciones de entrenamiento se van separando de forma recursiva creando las ramas del árbol hasta que cada observación queda aislada en un nodo terminal. Sin embargo, en los isolation tree, la selección de los puntos de división se hace de forma aleatoria. Aquellas observaciones con características distintas al resto, quedarán aisladas a las pocas divisiones, por lo que el número de nodos necesarios para llegar a estas observación desde el inicio del árbol (profundidad) es menor que para el resto.

```
#Cargamos el dataset y lo asignamos a la variable df
df = pd.read_csv('muestra1.csv')
#seleccionamos las columnas con las variables más predictoras. He hecho pruebas con distir
#ofrece mejores resultados
datos_X1 = pd.DataFrame(df[['Proto', 'SrcPt', 'DstPt', 'Flags', 'Tos']])
#definición de la variable objetivo
datos_y1 = pd.DataFrame(df[['class']])

#Tuneado de parámetros del algoritmo
modelo_isof1 = IsolationForest(
```

```
                n_estimators  = 1000, #número de árboles que forman el modelo
                max_samples   ='auto', #número de observaciones empleadas para entrenar ca
                contamination = 0.118, #proporción de anomalías esperadas en los datos de
                n_jobs        = -1,
                random_state  = 123, #semilla para garantizar la reproducibilidad de los r
            )
#Se procede a entrenar un modelo asumiendo que hay un 11,8% de observaciones anómalas en e
modelo_isof1.fit(X=datos_X1)
#generamos las predicciones y la asigamos a una variable
clasificacion_predicha1 = modelo_isof1.predict(X=datos_X1)
# Predicción valor anomalía
score_anomalia1 = modelo_isof1.score_samples(X=datos_X1)
#Calculamos el quantile o threshold de las anomalias a partir de las cuales, una observaci
cuantil_01 = np.quantile(score_anomalia1, q=0.118)
#Calculamos el score de anomalías para cada observación
score1 = pd.DataFrame(score_anomalia1)
#Creamos un dataset con los valores reales del dataset
anomalia1 = pd.DataFrame(datos_y1)
#Unimos los valores reales y las predicciones
df_resultados1 = score1.join(anomalia1)
#Renombramos las columnos por Score y Anomalia
df_resultados1 = df_resultados1.rename(columns={0:'score', 'class':'anomalia'})
#ordenamos en forma ascendiente los scores de anomalías
df_resultados1 = df_resultados1 \
                .sort_values('score', ascending=True) \
                .reset_index(drop=True)
#Convertimos el score anomalías en 1 (si es mayor que el quantil) o 0 (si es menor)
df_resultados1['clasificacion'] = np.where(df_resultados1.index <= 1237, 1, 0)
# Matriz de confusión de la clasificación final
pd.crosstab(
    df_resultados1.anomalia,
    df_resultados1.clasificacion
)
```

clasificacion	0	1
anomalia		
0	8131	1118
1	1117	120

Resultados también muy malos, con pocos aciertos en la clase de anomalías.

One-Class SVM

El algoritmo es una extensión natural del SVC (support vector classifier) para el caso de información no etiquetada. One-Class SVM busca capturar una región en donde se concentra la densidad de probabilidad, es decir encontrar una función que separe la gran mayoría de los datos hacia un lado (suministros normales), identificando de esta forma las “novedades”

(posibles suministros con irregularidades) como los datos que quedan del otro lado de la superficie de decisión.

```
#Cargamos el dataset
data = pd.read_csv('muestra1.csv')
#seleccionamos las variables más predictoras
df = data[['Duration', 'SrcPt', 'Packets', 'Tos', 'Packets_speed', 'Bytes_speed', 'DstPt']
#Definimos los parámetros del modelo
model = OneClassSVM(kernel = 'rbf', gamma = 0.25, nu = 0.5).fit(df)
#Realizamos las predicciones
y_pred = model.predict(df)
#creamos un dataframe con las predicciones generadas
y_pred = pd.DataFrame(y_pred)
#Si la predicción es -1 la pasamos a 1 (caso anomalía), sino la pasamos a 0 (caso normal)
y_pred.loc[y_pred[0]==1]=0
y_pred.loc[y_pred[0]==-1]=1
#definimos la variable objetivo
y_train = data[["class"]]
#Creamos matriz de confusión
confusion_matrix(y_train, y_pred)
```

```
array([[4272, 4977],
       [ 885,  352]], dtype=int64)
```

Resultados no muy buenos, pero se logra un aumento en la detección de anomalías

Autoencoder

Es un tipo de arquitectura de redes neuronales que pertenece al grupo de métodos de aprendizaje no supervisados. Esta arquitectura extrae las características más importantes del input eliminando el resto de poca relevancia. Esto permite que los autoencoders aprendan una representación de la información reducida siendo un perfecto método para comprimir información.

```
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model, Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from tensorflow.keras.losses import MeanSquaredLogarithmicError
from numpy.random import seed
seed(1)
```

```
#Cargamos dataset y seleccionamos las variables más predictoras
df = pd.read_csv('train_CIDDS.csv')
datos_X = pd.DataFrame(df[['Duration', 'Proto', 'DstPt', 'Bytes', 'Flags', 'Tos', 'Packets
datos_y = pd.DataFrame(df[['class']])
x_train = datos_X
y_train = datos_y

#cargamos el dataset de test
df1 = pd.read_csv("muestra1.csv")
x_test = pd.DataFrame(df1[['Duration', 'Proto', 'DstPt', 'Bytes', 'Flags', 'Tos', 'Packets
y_test = pd.DataFrame(df1[['class']])

# estandarizamos y hacemos fit
min_max_scaler = MinMaxScaler(feature_range=(0, 1))
x_train_scaled = min_max_scaler.fit_transform(x_train.copy())
x_test_scaled = min_max_scaler.transform(x_test.copy())

# creamos modelo en tensorflow
class AutoEncoder(Model):
  """
  Parameters
  ----------
  output_units: int
    Number of output units

  code_size: int
    Number of units in bottle neck
  """
#definimos el codificador con 512, 256 y 128 neuronas en cada capa
  def __init__(self, output_units, code_size=10):
    super().__init__()
    self.encoder = Sequential([
      Dense(512, activation='relu'),
      Dropout(0.1),
      Dense(256, activation='relu'),
      Dropout(0.1),
      Dense(128, activation='relu'),
      Dropout(0.1),
      Dense(code_size, activation='relu')
    ])
    #definimos el decodificador con 128, 256 y 512 neuronas en cada capa
    self.decoder = Sequential([
      Dense(128, activation='relu'),
      Dropout(0.1),
      Dense(256, activation='relu'),
      Dropout(0.1),
      Dense(512, activation='relu'),
      Dropout(0.1),
      Dense(output_units, activation='sigmoid')
    ])

  def call(self, inputs):
    encoded = self.encoder(inputs)
    decoded = self.decoder(encoded)
```

```
        return decoded
    #creamos el modelo con el algoritmo autoencoder
    model = AutoEncoder(output_units=x_train_scaled.shape[1])
    # configuramos el modelo
    model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer='adam')
    #hacemos un fit del modelo con los parámetros tuneados (se han probado epochs y batch_size
    history = model.fit(
        x_train_scaled,
        x_train_scaled,
        epochs=500,
        batch_size=128,
        validation_data=(x_test_scaled, x_test_scaled)
    )

    #funciones necesarias
    def find_threshold(model, x_train_scaled):
      reconstructions = model.predict(x_train_scaled)
      # provides losses of individual instances
      reconstruction_errors = tf.keras.losses.msle(reconstructions, x_train_scaled)
      # threshold for anomaly scores
      threshold = np.mean(reconstruction_errors.numpy()) \
          + np.std(reconstruction_errors.numpy())
      return threshold

    def get_predictions(model, x_test_scaled, threshold):
      predictions = model.predict(x_test_scaled)
      # provides losses of individual instances
      errors = tf.keras.losses.msle(predictions, x_test_scaled)
      # 1 = anomaly, 0 = normal
      anomaly_mask = pd.Series(errors) > threshold
      preds = anomaly_mask.map(lambda x: 1 if x == True else 0)
      return preds
    #obtenemos el valor del threshold
    threshold = find_threshold(model, x_train_scaled)
    print(f"Threshold: {threshold}")

    #generamos las predicciones
    predictions = get_predictions(model, x_test_scaled, threshold)
    predictions = pd.DataFrame(predictions) #convertimos en dataframe
    predictions = predictions.rename(columns={0:'class_pred'})#renombramos la columna 0 por cl
    y_test = y_test.rename(columns={1:'class'}) #renombramos la columna 1 por class
    y_test["class"] = y_test["class"].astype(str).astype(int)
    #calculamos accuracy del modelo
    accuracy_score(predictions, y_test)
    #calculamos la matriz de confusión
    from sklearn.metrics import confusion_matrix
    confusion_matrix(y_test, predictions)
```

```
    125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
    Epoch 472/500
    125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
    Epoch 473/500
    125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
    Epoch 474/500
    125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
    Epoch 475/500
    125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
```

```
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 476/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 477/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 478/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 479/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 480/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 481/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 482/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 483/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 484/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 485/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 486/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 487/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 488/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 489/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 490/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 491/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1946 - accuracy:
Epoch 492/500
125/125 [==============================] - 2s 18ms/step - loss: 0.1947 - accuracy:
Epoch 493/500
125/125 [==============================] - 2s 18ms/step - loss: 0.1948 - accuracy:
Epoch 494/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 495/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 496/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 497/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 498/500
125/125 [==============================] - 2s 17ms/step - loss: 0.1947 - accuracy:
Epoch 499/500
125/125 [==============================] - 2s 18ms/step - loss: 0.1947 - accuracy:
```

Muy buenos resultados. Se mejoran mucho con 500 epoch, aunque tarda bastante en generar las predicciones

```
df = pd.read_csv('train_CIDDS.csv')
datos_X = pd.DataFrame(df[['Duration', 'Proto', 'DstPt', 'Bytes', 'Flags', 'Tos', 'Packets
datos_y = pd.DataFrame(df[['class']])
```

```
x_train = datos_X
y_train = datos_y

df1 = pd.read_csv("muestra1.csv")
x_test = pd.DataFrame(df1[['Duration', 'Proto', 'DstPt', 'Bytes', 'Flags', 'Tos', 'Packets
y_test = pd.DataFrame(df1[['class']])

min_max_scaler = MinMaxScaler(feature_range=(0, 1))
x_train_scaled = min_max_scaler.fit_transform(x_train.copy())
x_test_scaled = min_max_scaler.transform(x_test.copy())

#funciones necesarias
def find_threshold(model, x_train_scaled):
  reconstructions = model.predict(x_train_scaled)
  # provides losses of individual instances
  reconstruction_errors = tf.keras.losses.msle(reconstructions, x_train_scaled)
  # threshold for anomaly scores
  threshold = np.mean(reconstruction_errors.numpy()) \
      + np.std(reconstruction_errors.numpy())
  return threshold

def get_predictions(model, x_test_scaled, threshold):
  predictions = model.predict(x_test_scaled)
  # provides losses of individual instances
  errors = tf.keras.losses.msle(predictions, x_test_scaled)
  # 1 = anomaly, 0 = normal
  anomaly_mask = pd.Series(errors) > threshold
  preds = anomaly_mask.map(lambda x: 1 if x == True else 0)
  return preds


import joblib
autoencoder_model3 = open('autoencoder_model3.pkl','rb')
model = joblib.load(autoencoder_model3)

df = pd.read_csv('muestra1.csv')
#df = df.iloc[10483:10484]
X_test = pd.DataFrame(df[['Duration', 'Proto', 'DstPt', 'Bytes', 'Flags', 'Tos', 'Packets_
min_max_scaler = MinMaxScaler(feature_range=(0, 1))
x_test_scaled = min_max_scaler.fit_transform(X_test.copy())
x_test_scaled = min_max_scaler.transform(X_test.copy())
Y_test = pd.DataFrame(df[['class']])
#result = model1.score(X_test, Y_test)
threshold = find_threshold(model, x_train_scaled)

predictions = model.predict(x_test_scaled)
errors = tf.keras.losses.msle(predictions, x_test_scaled)
anomaly_mask = pd.Series(errors) > threshold
preds = anomaly_mask.map(lambda x: 1 if x == True else 0)

predictions = get_predictions(model, x_test_scaled, threshold)
#predictions = pd.DataFrame(predictions)
#predictions = predictions.rename(columns={0:'class_pred'})
#y_test = y_test.rename(columns={1:'class'})
#y_test["class"] = y_test["class"].astype(str).astype(int)
```

```
#accuracy_score(predictions, y_test)

#predictions = get_predictions(model3, x_test_scaled, threshold)
if predictions[0] == 1:
  print("Test traffic flow predicted to be Anomalous")
else:
  print("Test traffic flow predicted to be Normal")
```

```
Test traffic flow predicted to be Normal
```

11 min 40 s terminée à 17:45