ESCUELA POLITÉCNICA SUPERIOR DE CÓRDOBA
Universidad de Córdoba

EPSC

TRABAJO FIN DE GRADO
Grado en Ingeniería Informática

Estudio comparativo de métodos de aprendizaje automático en la detección de malware

Autor: Manuel Jesús Mariscal Romero
Directores: D. David Guijo Rubio
D. Víctor Manuel Vargas Yun

julio, 2025

UNIVERSIDAD D CÓRDOBA

Resumen

rellenar
Palabras clave: Palabra-1, Palabra-2, Palabra-3, Palabra-4.

Abstract

rellenar
Keywords: Palabra-1, Palabra-2, Palabra-3, Palabra-4.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Capítulo 2

Estado de la técnica

2.1. Aprendizaje automático

2.1.1. Balanceo de datos

2.1.1.1. Sobremuestreo

2.1.1.2. Submuestreo

2.1.2. Reducción de la dimensionalidad

2.1.2.1. Análisis de componentes principales

2.1.2.2. Análisis factorial

2.1.2.3. Descomposición en valores singulares

2.2. Ciberseguridad

2.2.1. *Malware*

Capítulo 3

Formunación del problema

Capítulo 4

Objetivos

Debe de ser igual a los mencionados en el anteproyecto.

Describir el objetivo principal y los objetivos especificos llevados a cabo para conseguir el objetivo principal.

Capítulo 5

Metodología de trabajo

Capítulo 6

Desarrollo y experimentación

6.1. Selección del conjunto de datos

En lo que a *malware* se refiere, *BODMAS* [1] es uno de los conjuntos de datos más completos en la actualidad, con la ventaja para este proyecto de ya estar procesado y tener una amplia bibliografía. Otra opción interesante puede ser *VirusShare* [2], ya que cuenta con más de 99 millones de muestras de *malware* actualizadas pero tiene varios inconvenientes para este proyecto. El primero es que no incluye muestras de *software* no malicioso y el segundo que necesita un procesamiento previo para extraer las características. Todo esto conlleva un aumento considerable para la realización del proyecto. Otra de las opciones estudiadas ha sido *theZoo* [3]. En cuanto a este repositorio hemos podido observar que tiene los mismos inconvenientes que *VirusShare* y no tiene sus ventajas. Por último tenemos *Microsoft Malware Classification* [4]. En este caso tenemos un conjunto de datos muy amplio con casi medio *terabyte*, pero además de los inconvenientes ya comentados en los anteriores conjuntos, solo incluye *malware* que afecta a equipos *Windows*, lo que limitaría considerablemente el alcance del estudio.

Teniendo en cuenta todo lo comentado hasta ahora sobre los distintos conjuntos de datos considerados, hemos decidido usar *BODMAS*, ya que es el que mejor se adapta a las necesidades del estudio

6.2. Procesamiento del conjunto de datos

Dadas las limitaciones *hardware* y la cantidad de datos, aproximadamente 135000 patrones y 2400 atributos por cada patrón, es necesario hacer un procesamiento previo del conjunto de datos. Para ello hemos tenido en cuenta varios enfoques. Por un lado, *BODMAS* nos permite hacer una distinción entre clasificación binaria y clasificación multiclase, pero para ello es necesario reordenar los datos, ya que se encuentran distribuidos en varios archivos. Por otro lado, es necesario reducir la cantidad de datos. A continuación veremos los distintos enfoques.

6.2.1. Clasificación multiclase

El conjunto de datos seleccionado se divide en varios archivos:

- *bodmas.npz*: incluye la matriz de patrones de entrada en formato de matriz de *python* y la matriz de salidas deseadas.
- *bodmas_metadata.csv*: la información relevante para nuestro problema es la columna *sha* que contiene la función *hash* de todo el conjunto de datos.
- *bodmas_malware_category.csv*: contiene la función *hash* del *malware* y la categoría a la que pertenece.

Dado que las distintas categorías se encuentran en formato texto, es necesario codificarlas para poder trabajar con ellas. La codificación elegida ha sido la representada en la tabla 6.1.

Para obtener una nueva matriz de salidas deseadas que incluya los tipos de *malware*, una vez cargados los datos en sus correspondiente variables de *python*, usamos la función *merge* [5] perteneciente a la clase *pandas.DataFrame* para incluir en *metadata* los datos de *mw_category['category']* en las entradas donde coincide la columna *sha*.

Antes de codificar necesitamos darle una etiqueta a los datos vacíos, los cuales significan que esa muestra es benigna. Para ello usamos la función *pandas.DataFrame.fillna* [6], que nos permite completar datos vacíos de distintas formas. Para nuestro caso usamos la etiqueta *benign*. También eliminamos las columnas que no vamos a necesitar, dejando solo la categoría a la que pertenece cada muestra.

Ahora podemos codificar los datos usando la función *pandas.DataFrame.map* [7]. Este método aplica una función que acepta y devuelve un valor escalar a cada elemento del DataFrame, lo que permite asignar un valor numérico a cada clase.

Tabla 6.1: Codificación de las clases *malware*.

Categoría	Codificación	Nº de patrones
benign	0	77142
trojan	1	29972
worm	2	16697
backdoor	3	7331
downloader	4	1031
informationstealer	5	448
dropper	6	715
ransomware	7	821
rootkit	8	3
cryptominer	9	20
pua	10	29
exploit	11	12
virus	12	192
p2p-worm	13	16
trojan-gamethief	14	6

El código utilizado para esta tarea se encuentra en el Anexo A.1.

6.2.2. Reducción del conjunto de datos

Reducir el número de datos con el que vamos a trabajar tiene el objetivo de principal de disminuir el tiempo que los algoritmos van a necesitar para procesar la información sin perjudicar la integridad de los datos, ya que los resultados del estudio podrían verse afectados y llevar a unas conclusiones erróneas. Esta tarea se puede enfrentar desde dos planteamientos distintos: condensar el número de patrones o el número de características. Ambos planteamientos se han estudiado de forma teórica en esta memoria en las secciones 2.1.1 y 2.1.2 respectivamente. Las técnicas elegidas son *undersampling* por simplicidad y *PCA* porque según el estudio *A Low Complexity ML-Based Methods for Malware Classification* [8] se obtienen unos resultados algo más precisos que con otros métodos.

El código utilizado se encuentra en el anexo A.2. A continuación se explicarán los pasos seguidos.

6.2.2.1. Número de patrones

Como ya hemos estudiado en la sección 2.1.1.2, el submuestreo o *undersampling* en inglés, es una técnica para abordar el desbalance de clases en un conjunto de datos,

especialmente cuando una de las clases tiene muchos más patrones que la otra. En nuestro caso, el desbalance no es demasiado grande ya que *BODMAS* contiene 57293 muestras *malware* y 77142 muestras benignas.

El método *RandomUnderSampler* [9] de la librería *Imbalanced learn* nos permite varias formas de actuar, siendo la que nos interesa para este estudio la que nos permite elegir manualmente el número de patrones de cada clase. Hemos elegido una cantidad de 15000 patrones en por clase.

6.2.2.2. Número de características

Este método, también conocido como reducción de la dimensionalidad, consiste en reducir el número de variables de las que consta el problema. Para aplicar el método matemático-estadístico de análisis de componentes principales, *PCA* por sus siglas en inglés, usamos la clase *PCA* [10] perteneciente a *sklearn.decomposition*. Esta clase nos permite entrenar el modelo y transformar el conjunto de datos tanto para el conjunto de entrenamiento como para el de test. Para ello será necesario separar previamente los datos, ya que *BODMAS* no cuenta con esta división.

6.2.2.3. Elección final del nuevo conjunto de datos

Para poder decidir como será el conjunto de entrenamiento final se han hecho distintos conjuntos de datos sobre los que se probarán algunos algoritmos. Los conjuntos son los siguientes:

- Clasificación binaria con *PCA*.
- Clasificación binaria con *PCA* y *Undersampling* con 15000 patrones por clase.
- Clasificación multiclase con *PCA*.

Los resultados obtenidos se reflejan en las tablas 6.2, 6.3 y 6.4 respectivamente.

Tabla 6.2: Clasificación binaria con *PCA*.

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission tree	0.885	1.000	1.000	1.000	0.972	0.971	1.000
Random forest	25.91	1.000	1.000	1.000	0.984	0.976	1.000
K-NN	0.095	0.973	0.970	1.000	0.963	0.963	1.000

En cuanto a la clasificación binaria, hemos decidido usar el conjunto de datos en el que se ha aplicado tanto *PCA* como *undersampling*, ya que, aunque los resultados

Tabla 6.3: Clasificación binaria con *PCA* y *Undersampling*.

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission tree	0.184	1.000	1.000	1.000	0.945	0.936	1.000
Random forest	4.926	1.000	1.000	1.000	0.963	0.957	1.000
K-NN	0.016	0.954	0.948	1.000	0.938	0.931	1.000

Tabla 6.4: Clasificación multiclase con *PCA*

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission tree	1.059	0.999	0.895	0.999	0.939	0.000	0.976
Random forest	30.04	0.999	0.895	0.999	0.955	0.000	0.981
K-NN	0.088	0.951	0.000	0.981	0.936	0.000	0.975

son similares en ambos conjuntos, el tiempo es considerablemente más bajo y dadas las limitaciones del equipo disponible puede ser beneficioso a la hora de probar algoritmos más complejos.

Para la clasificación multiclase hay varios métodos que podemos usar para reducir el tamaño del conjunto de datos, como el *clustering* o variantes del método de *undersampling* ya utilizado en clasificación binaria. A pesar de ello, estos métodos tienen una mayor complejidad de aplicación y la reducción de las dimensiones no es el objeto de este estudio. Por otro lado, esta decisión puede suponer algunos problemas al usar técnicas como *GridSearchCV* o la validación cruzada, ya que incrementan considerablemente el tiempo de entrenamiento.

También en referencia a la clasificación multiclase, podemos ver en la tabla 6.4 que la métrica de mínima sensibilidad es 0 para todos los casos de test. Como ya se ha explicado en esta memoria, mide cómo de bien se clasifica la clase peor clasificada y un valor de 0 indica que alguna de las clases no se ha clasificado bien. Como podemos ver en la matriz de confusión representada en la imagen 6.1, algunas de las clases con menos patrones tienen dificultades para obtener una buena clasificación debido a la falta de información en el entrenamiento. Algunos clasificadores tienen la opción de asignar un peso a los patrones de cada clase inversamente proporcional al número de patrones de la clase, de manera que todas las clases tengan el mismo peso en el entrenamiento, pero no se consiguen mejores resultados.

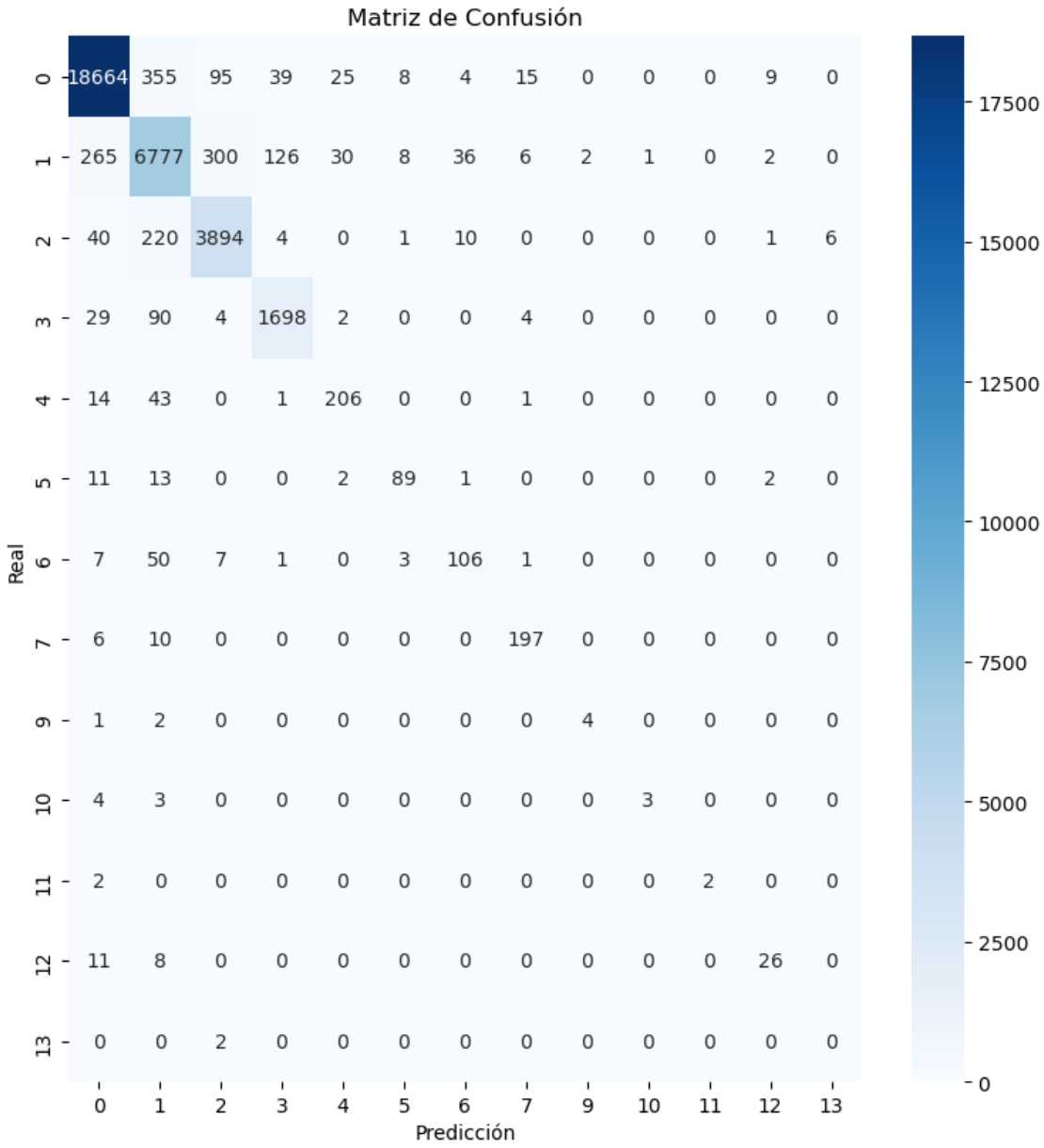

Figura 6.1: Título de la figura para el texto

Según el estudio *Malware Behavior Analysis: Learning and Understanding Current Malware Threats* [11], algunos de los tipos de *malware* que tenemos con menos patrones, se pueden agrupar en algunas de las clases más representadas de nuestro conjunto de datos. En este estudio se comenta que *p2p-worm* añade un comportamiento específico al comportamiento de un gusano, generando problemas de red y de pérdida de datos. Algo similar pasa con *Gamethief trojan*. De esta forma podemos agrupar estos patrones a sus respectivas clases similares sin perder efectividad a la hora de clasificar y además eliminar así dos de las clases que nos pueden dar problemas por falta de información.

Por otro lado, se han planteado dos formas de solucionar este problema, aunque ambas presentan inconvenientes:

- Eliminar las clases menos representadas. Tiene el riesgo de no reconocer un nuevo patrón si es de un tipo distinto de *malware*.
- Agruparlas en una nueva clase que represente varios tipos de *malware*. En este caso estamos suponiendo que los patrones agrupados tienen unas características similares.

Finalmente hemos decidido agrupar las clases con menos de 30 patrones en una nueva categoría *otros*. Por número de patrones sería recomendable agrupar también la clase *virus*, pero podría tener demasiado peso en la categoría otros y hemos considerado que es lo suficientemente relevante como para estudiarla por separado. En la tabla 6.6 podemos ver que, aunque mejoramos la mínima sensibilidad, no se producen unas mejoras significativas en la precisión de clasificación. Esto podría deberse a que cuando agrupamos distintas clases con pocos patrones, no se gane capacidad de clasificación en clases mayoritarias. Podemos ver la nueva codificación en la tabla 6.5

Por último, se han considerado otras opciones para mejorar la clasificación de las clases minoritarias, pero podrían exceder la complejidad de este proyecto:

- Utilizar métodos de sobremuestreo, ya mencionados en la sección 2.1.1.1, que consisten en aumentar la cantidad de patrones de estas clases de forma sintética.
- Utilizar métodos jerárquicos que primero clasifiquen usando la categoría *otros*, para después dividirla en sus diferentes clases y entrenar un modelo específico.

Tabla 6.5: Nueva codificación de las clases *malware*.

Categoría	Codificación	Nº de patrones
benign	0	77142
trojan	1	29978
worm	2	16713
backdoor	3	7331
downloader	4	1031
informationstealer	5	448
dropper	6	715
ransomware	7	821
virus	8	192
otros	9	64

Tabla 6.6: Clasificación multiclase con la nueva codificación.

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission tree	1.220	0.998	0.992	0.998	0.938	0.670	0.975
Random forest	31.201	0.998	0.993	0.998	0.953	0.670	0.980
K-NN	0.083	0.951	0.431	0.980	0.936	0.333	0.974

6.3. Experimentación

6.3.1. Detección de *malware*

Tabla 6.7: caption

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	1.000	1.000	1.000	0.951	0.942	1.000
1	1.000	1.000	1.000	0.944	0.935	1.000
2	1.000	1.000	1.000	0.942	0.935	1.000
3	1.000	1.000	1.000	0.948	0.938	1.000
4	1.000	1.000	1.000	0.953	0.946	1.000
5	0.998	0.997	1.000	0.947	0.936	1.000
6	0.998	0.997	1.000	0.947	0.941	1.000
7	1.000	1.000	1.000	0.949	0.946	1.000
8	0.999	0.998	1.000	0.948	0.937	1.000
9	1.000	1.000	1.000	0.950	0.940	1.000
Mean	0.999	0.999	1.000	0.948	0.940	1.000
STD	0.001	0.001	0.000	0.003	0.004	0.000

Capítulo 7

Resultados y discusión

Capítulo 8

Conclusiones y recomendaciones

Bibliografía

[1] Bodmas. URL https://whyisyoung.github.io/BODMAS/.

[2] Virusshare. URL https://virusshare.com/.

[3] thezoo. URL https://github.com/ytisf/theZoo.

[4] Microsoft malware classification challenge. URL https://www.kaggle.com/c/malware-classification/data.

[5] pandas.dataframe.merge. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html.

[6] pandas.dataframe.fillna. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html.

[7] pandas.dataframe.map. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.map.html.

[8] A low complexity ml-based methods for malware classification. URL https://www.researchgate.net/publication/383827671_A_Low_Complexity_ML-Based_Methods_for_Malware_Classification.

[9] Randomundersampler. URL https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html.

[10] sklearn.decomposition.pca. URL https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[11] Mohamad Zolkipli and Aman Jantan. Malware behavior analysis: Learning and understanding current malware threats. URL https://www.researchgate.net/publication/232657598_Malware_Behavior_Analysis_Learning_and_Understanding_Current_Malware_Threats.

Anexo A

Código del programa

A.1. Codificación de las categorías *malware*

```
X, y        = load('bodmas/bodmas.npz')
metadata    = pd.read_csv('bodmas/bodmas_metadata.csv')
mw_category = pd.read_csv('bodmas/bodmas_malware_category.csv')

# Incluimos los valores de 'category' en metadata cuando coinciden
   los valoes de 'sha'
mw_category = metadata.merge(mw_category, on = 'sha', how = 'left')

# Rellenamos los huecos como software benigno
mw_category['category'] = mw_category['category'].fillna('benign')

# Eliminamos todas las columnas excepto 'category'
mw_category = mw_category['category']

# Codificamos las categorias de malware
category = {
  'benign': 0, 'trojan': 1, 'worm': 2, 'backdoor': 3,
  'downloader': 4, 'informationstealer': 5, 'dropper': 6,
  'ransomware': 7, 'rootkit': 8, 'cryptominer': 9, 'pua': 10,
  'exploit': 11, 'virus': 12, 'p2p-worm': 13, 'trojan-gamethief':
   14
}

mw_category = mw_category.map(category)

y = mw_category.to_numpy()

save('bodmas/bodmas_multiclass.npz', X, y)
```

A.2. Reducción de la dimensionalidad

```
def resampling(X, y, n_components = 5, size = 15000, u = False):
  if u:
    rus  = RandomUnderSampler(sampling_strategy = {0: size, 1: size
   })
    # rus  = RandomUnderSampler(sampling_strategy = 'majority')
    X, y = rus.fit_resample(X, y)

  X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.25, random_state = 1
  )

  pca     = PCA(n_components)
  X_train = pca.fit_transform(X_train)
  X_test  = pca.transform(X_test)

  return X_train, X_test, y_train, y_test
```

A.3. Pruebas para la elección del conjunto de datos

```
file = {'pca_binary', 'resampling_binary', 'pca_multiclass'}
clf  = None

print('clasificador,dataset,n patrones,n caracteristicas,accuracy,
    tiempo')

for i in range(3):
  if i == 0: clf = DecisionTreeClassifier()
  elif i == 1: clf = RandomForestClassifier()
  else: clf = KNeighborsClassifier()

  for train_file in file:

    X_train, y_train = load('bodmas/' + train_file + '_train.npz')
    X_test, y_test = load('bodmas/' + train_file + '_test.npz')

    # Entrenar el modelo
    inicio = time.time()
    clf.fit(X_train, y_train)
    tiempo = time.time() - inicio

    # Predecir sobre el conjunto de prueba
    y_pred = clf.predict(X_test)

    # Evaluar
    accuracy = accuracy_score(y_test, y_pred)

    print(f'{i},{train_file},{X_train.shape},{accuracy:.3f},{tiempo
    :.3f}')
```