

PROYECTO INFORME FINAL

PROYECTO DE ANALÍTICA DE DATOS



**UNIVERSIDAD
DE ANTIOQUIA**

1 8 0 3

INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

**PRESENTADO POR:
SAMUEL GIL ARBOLEDA**

UNIVERSIDAD DE ANTIOQUIA

**DOCENTE:
RAÚL RAMOS POLLÁN
2022-1**

1. INTRODUCCIÓN

1.1 PROBLEMA PREDICTIVO

En este trabajo se aborda el problema de clasificación, que tiene como objetivo predecir si un sitio web es considerado malicioso (phishing) o no, teniendo en cuenta características en la URL que normalmente tienen estos sitios. Este problema es de vital importancia debido a que estos sitios son difíciles de reconocer para gente que no tiene conocimientos en tecnología, por lo cual es necesario resolverlo mediante técnicas de aprendizaje de máquina debido a que reconocerlas manualmente podría conllevar mucho tiempo o ser un trabajo imposible, ya que la cantidad de páginas web existentes son prácticamente infinitas.

1.2 DATASET

El dataset a utilizar proviene de un repositorio de datos de machine learning llamado UCI (<https://archive.ics.uci.edu/ml/datasets/phishing%20websites#>). La base de datos de este proyecto es "Phishing Websites Data Set" [5], cuenta con 11055 muestras, y 31 variables, las cuales 30 son para explicar las características de una url de un sitio web, y la otra es la variable de salida "Result" la cual se encarga de decirnos si un sitio web es phishing o no.

Las variables presentes en el dataset pueden contener 2 o 3 de los siguientes valores: -1, 0 ó 1, lo cual significa que cumplen con la condición propuesta en la variable, es sospechosa, y no la cumple, respectivamente. Además no hay datos faltantes en ninguna característica.

#	Nombre variable	Codificación
0	having_IP_Address	(categorical - signed numeric) : { -1,1 }
1	URL_Length	(categorical - signed numeric) : { 1,0,-1 }
2	Shortining_Service	(categorical - signed numeric) : { 1,-1 }
3	having_At_Symbol	(categorical - signed numeric) : { 1,-1 }
4	double_slash_redire cting	(categorical - signed numeric) : { -1,1 }
5	PrefixSuffix-	(categorical - signed numeric) : { -1,1 }
6	having_Sub_Domain	(categorical - signed numeric) : { -1,0,1 }
7	SSLfinal_State	(categorical - signed numeric) : { -1,1,0 }
8	Domain_registeratio n_length	(categorical - signed numeric) : { -1,1 }
9	Favicon	(categorical - signed numeric) : { 1,-1 }

10	port	(categorical - signed numeric) : { 1,-1 }
11	HTTPS_token	(categorical - signed numeric) : { -1,1 }
12	RequestURL	(categorical - signed numeric) : { 1,-1 }
13	URL_of_Anchor	(categorical - signed numeric) : { -1,0,1 }
14	Links_in_tags	(categorical - signed numeric) : { 1,-1,0 }
15	ServerFormHandler	(categorical - signed numeric) : { -1,1,0 }
16	Submitting_to_email	(categorical - signed numeric) : { -1,1 }
17	AbnormalURL	(categorical - signed numeric) : { -1,1 }
18	Redirect	(categorical - signed numeric) : { 0,1 }
19	on_mouseover	(categorical - signed numeric) : { 1,-1 }
20	RightClick	(categorical - signed numeric) : { 1,-1 }
21	popUpWidnow	(categorical - signed numeric) : { 1,-1 }
22	Iframe	(categorical - signed numeric) : { 1,-1 }
23	AgeofDomain	(categorical - signed numeric) : { -1,1 }
24	DNSRecord	(categorical - signed numeric) : { -1,1 }
25	web_traffic	(categorical - signed numeric) : { -1,0,1 }
26	PageRank	(categorical - signed numeric) : { -1,1 }
27	GoogleIndex	(categorical - signed numeric) : { 1,-1 }
28	LinksPointingToPage	(categorical - signed numeric) : { 1,0,-1 }
29	Statistical_report	(categorical - signed numeric) : { -1,1 }
30	Result	(categorical - signed numeric) : { -1,1 }

Fig 1. Columnas del dataset

1.3 MÉTRICAS

Para evaluar el sistema se van a usar las siguientes métricas de evaluación: accuracy y f1 score. Siendo accuracy la medida principal.

Por otra parte, en cuanto a la métrica de negocio, se tiene interés en que las predicciones sean lo suficientemente confiables para saber si un sitio web es de phishing o no. Con esta información un navegador de internet podría evitar que sus usuarios entren a estos sitios maliciosos solo leyendo la página a la cual se dirigen.

1.4 DESEMPEÑO ESPERADO

Lo que se esperaría de un modelo de este tipo es obtener una predicción con bastante desempeño (más de un 80% de precisión) , porque no sería viable tener muchos falsos positivos, ya que bloquear constantemente las páginas de un usuario de un navegador podría hacer que este deje de usarlo. En un ambiente productivo sería usado como filtro para evitar que los usuarios entren a las páginas que sean sospechosas.

2. EXPLORACIÓN DESCRIPTIVA DEL DATASET

La base de datos de este proyecto es “Phishing Websites Data Set”[1], cuenta con 11055 muestras, y 31 variables, las cuales 30 son para explicar las características de una url de un sitio web, y la otra es la variable de salida “Result” la cual se encarga de decirnos si un sitio web es phishing o no. La distribución de las clases es de 6157 muestras para la clase 1 (no phishing) y 4898 muestras para la clase -1 (phishing), por lo cual es un problema balanceado al no tener una diferencia significativa entre el número de muestras de una clase con respecto a la otra.

Para evaluar el sistema se usaron las siguientes medidas de evaluación: accuracy y f1 score. Siendo accuracy la medida principal.

Se procedió entonces en una primera instancia a realizar la respectiva exploración y limpieza de los datos del dataset:

Out []:

	having_IP_Address	URL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix_Suffix	having_Sub_Domain	SSLfinal_State	Domain_registration_length	F
0	-1	1	1	1	-1	-1	-1	-1	-1	
1	1	1	1	1	1	-1	0	1	-1	
2	1	0	1	1	1	-1	-1	-1	-1	
3	1	0	1	1	1	-1	-1	-1	1	
4	1	0	-1	1	1	-1	1	1	-1	

5 rows x 31 columns

Dimensiones del dataset:

In []:

data.shape

Out []:

(11055, 31)

Fig 2. Dataframe del dataset

En las imágenes que se muestran a continuación, se evidencia que no hay presencia de datos nulos o faltantes. **Nota:** Si bien para el proyecto se pedía una un porcentaje específico de datos nulos, por temas de tiempo no se pudieron simular correctamente.

```
In [ ]: data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11055 entries, 0 to 11054
Data columns (total 31 columns):
#   Column                                Non-Null Count  Dtype
---  ---                                ---
0   having_IP_Address                    11055 non-null  int64
1   URL_Length                          11055 non-null  int64
2   Shortining_Service                  11055 non-null  int64
3   having_At_Symbol                    11055 non-null  int64
4   double_slash_redirecting            11055 non-null  int64
5   Prefix_Suffix                      11055 non-null  int64
6   having_Sub_Domain                  11055 non-null  int64
7   SSLfinal_State                     11055 non-null  int64
8   Domain_registration_length          11055 non-null  int64
9   Favicon                            11055 non-null  int64
10  port                               11055 non-null  int64
11  HTTPS_token                        11055 non-null  int64
12  Request_URL                        11055 non-null  int64
13  URL_of_Anchor                      11055 non-null  int64
14  Links_in_tags                      11055 non-null  int64
15  SFH                                11055 non-null  int64
16  Submitting_to_email                11055 non-null  int64
17  Abnormal_URL                       11055 non-null  int64
18  Redirect                           11055 non-null  int64
19  on_mouseover                       11055 non-null  int64
20  RightClick                         11055 non-null  int64
21  popUpWidnow                        11055 non-null  int64
22  Iframe                             11055 non-null  int64
23  age_of_domain                      11055 non-null  int64
24  DNSRecord                          11055 non-null  int64
25  web_traffic                        11055 non-null  int64
26  Page_Rank                          11055 non-null  int64
27  Google_Index                       11055 non-null  int64
28  Links_pointing_to_page              11055 non-null  int64
29  Statistical_report                  11055 non-null  int64
30  Result                             11055 non-null  int64
dtypes: int64(31)
memory usage: 2.6 MB
```

Fig 3. Datos nulos

```
Out[ ]: having_IP_Address      0
        URL_Length            0
        Shortining_Service     0
        having_At_Symbol       0
        double_slash_redirecting 0
        Prefix_Suffix          0
        having_Sub_Domain      0
        SSLfinal_State         0
        Domain_registration_length 0
        Favicon                0
        port                   0
        HTTPS_token            0
        Request_URL            0
        URL_of_Anchor          0
        Links_in_tags          0
        SFH                    0
        Submitting_to_email    0
        Abnormal_URL           0
        Redirect               0
        on_mouseover           0
        RightClick             0
        popUpWidnow           0
        Iframe                 0
        age_of_domain          0
        DNSRecord              0
        web_traffic            0
        Page_Rank              0
        Google_Index           0
        Links_pointing_to_page 0
        Statistical_report      0
        Result                 0
dtype: int64
```

Fig 4. Datos faltantes

La explicación de cada columna se encuentra en la documentación del dataset en el siguiente link: <https://archive.ics.uci.edu/ml/machine-learning-databases/00327/Phishing%20Web%20sites%20Features.docx>, es bastante extensa, por lo que no se añade a este

documento.

El nombre de las columnas presentes en el dataset son las siguientes:

```
In [ ]: data.columns

Out[ ]: Index(['having_IP_Address', 'URL_Length', 'Shortining_Service',
              'having_At_Symbol', 'double_slash_redirecting', 'Prefix_Suffix',
              'having_Sub_Domain', 'SSLfinal_State', 'Domain_registration_length',
              'Favicon', 'port', 'HTTPS_token', 'Request_URL', 'URL_of_Anchor',
              'Links_in_tags', 'SFH', 'Submitting_to_email', 'Abnormal_URL',
              'Redirect', 'on_mouseover', 'RightClick', 'popUpWidnow', 'Iframe',
              'age_of_domain', 'DNSRecord', 'web_traffic', 'Page_Rank',
              'Google_Index', 'Links_pointing_to_page', 'Statistical_report',
              'Result'],
              dtype='object')
```

Fig 5. Columnas del dataset

Adicionalmente, se decidió eliminar la columna 'ID' ya que esta no es una variable que sea relevante para el análisis.

```
In [ ]: del data["id"]
        data.head()
```

Fig 6. Eliminación columna 'ID'

2.1 NORMALIZACIÓN, BALANCEO DE LOS DATOS y TRAIN TEST SPLIT

En la exploración del dataset, se evidenció que no existe un desbalance en los datos con un total de 11055 muestras, 6157 muestras para la clase 1 (no phishing) y 4898 muestras para la clase -1 (phishing).

```
Out[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa68b6e71d0>
```

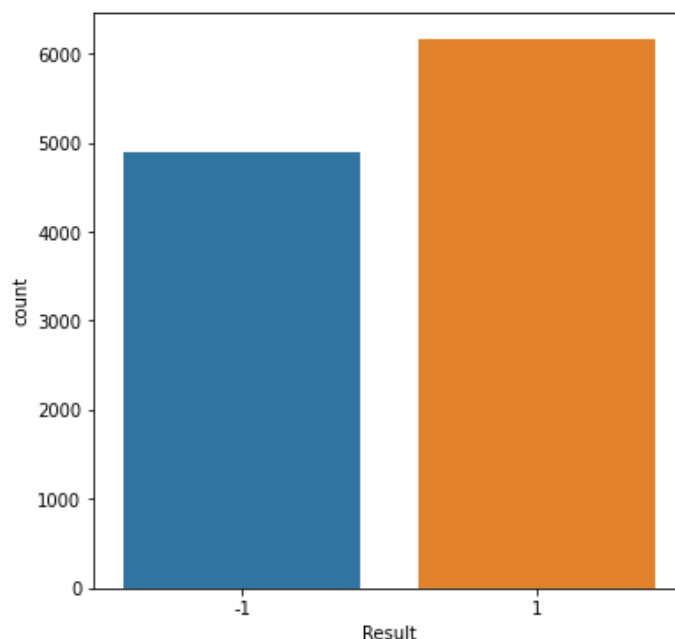


Fig 7. balance en los datos

Se dividió el dataset en dos bloques destinados al entrenamiento de los datos y validación del modelo:

```
Separacion de X y Y:

In [ ]: X = data.drop('Result', axis=1).values
        Y = data['Result'].values
        print (X.shape , Y.shape)
        #1

(11055, 30) (11055,)

Separacion de test y train: (Para probar codigos y disminuir tiempo de ejecucion)

In [ ]: #Para pruebas con modelos complicados
        from sklearn.model_selection import train_test_split
        X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.01, random_state=0)
        print (X_test.shape , Y_test.shape)

(111, 30) (111,)
```

Fig 8. Train Test Split

3. ALGORITMOS DE PREDICCIÓN

Se seleccionaron los algoritmos: Decision Tree y Random Forest para realizar el respectivo entrenamiento de los datos y consecuentemente generar modelos predictivos.

3.1 HIPER PARÁMETROS PARA DOS ALGORITMOS PREDICTIVOS

Con la finalidad de encontrar los mejores hiper parámetros para los algoritmos predictivos seleccionados: Decision Tree y Random Forest, se procedió a iterar en cada fold el hiper parámetro a analizar para así obtener la eficiencia de entrenamiento y de test de cada uno. Adicionalmente, se empleó el método de validación cruzada K-Fold para distribuir los datos de testeo y entrenamiento de manera equitativa.

3.1.1 DECISION TREE

Para este algoritmo se usó el hiper parámetro de profundidad del árbol, se usaron distintas profundidades del árbol (3,10,20 y 50 veces).

En la siguiente imagen se logra ver el código empleado para hallar los mejores hiper parámetros en el algoritmo Decision Tree:

```

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeClassifier
def algoritmo_dt(depths,X, Y):
    """función que realiza experimentos de árboles de decisión
    depths: list[int] lista con la profundidad de árboles a experimentar
    normalize bool: Indica si se aplica normalización a los datos
    X: matriz con las características
    Y: matriz de numpy con etiquetas
    retorna: dataframe con:
        - profundidad de los árboles
        - eficiencia con datos de entrenamiento
        - desviación estándar con datos de entrenamiento
        - eficiencia con datos de prueba
        - desviación estándar con datos de prueba
    """
    folds = 4
    skf = KFold(n_splits=folds)
    resultados = pd.DataFrame()
    idx = 0
    for depth in depths:
        # para almacenar los errores intermedios
        EficienciaTrain = []
        EficienciaVal = []
        for train, test in skf.split(X, Y):
            Xtrain = X[train,:]
            Ytrain = Y[train]
            Xtest = X[test,:]
            Ytest = Y[test]

            #Se llama la función para crear y entrenar el modelo usando los datos de entrenamiento
            modelo = DecisionTreeClassifier(max_depth=depth)
            modelo = modelo.fit(Xtrain, Ytrain)
            #Se predice con muestras de entrenamiento
            Ytrain_pred = modelo.predict(Xtrain)
            #Se predice con muestras de pruebas
            Ytest = modelo.predict(Xtest)
            #Evaluamos las predicciones del modelo con los datos de test
            EficienciaTrain.append(np.mean(Ytrain_pred.ravel() == Ytrain.ravel()))
            EficienciaVal.append(np.mean(Ytest.ravel() == Ytest.ravel()))

        resultados.loc[idx,'Profundidad del árbol'] = depth
        resultados.loc[idx,'Eficiencia con datos de entrenamiento'] = np.mean(EficienciaTrain)
        resultados.loc[idx,'Desviación estándar con datos de entrenamiento'] = np.std(EficienciaTrain)
        resultados.loc[idx,'Eficiencia con datos de prueba'] = np.mean(EficienciaVal)
        resultados.loc[idx,'Desviación estándar con datos de prueba'] = np.std(EficienciaVal)
        idx= idx +1

    return (resultados)

```

Fig 9. Código hiper parámetros DT.

Se puede observar que para el algoritmo Decision Tree los mejores resultados se obtienen con una profundidad de árbol de 50.

```
[ ] resultados_dt = experimentar_dt([3,10,20,50],MinMaxScaler().fit_transform(X), Y)
```

```
resultados_dt
```

	profundidad del arbol	eficiencia de entrenamiento	desviacion estandar entrenamiento	eficiencia de prueba	desviacion estandar prueba	accuracy	f1_score
0	3.0	0.908227	0.001163	0.904029	0.011213	0.904029	0.915764
1	10.0	0.958591	0.001634	0.943825	0.006790	0.943825	0.949381
2	20.0	0.989326	0.000761	0.962272	0.019184	0.962272	0.966199
3	50.0	0.989889	0.000318	0.962543	0.019835	0.962543	0.966425

Fig 10. Resultados - DT

Después se realiza la curva de aprendizaje, dando el siguiente resultado:

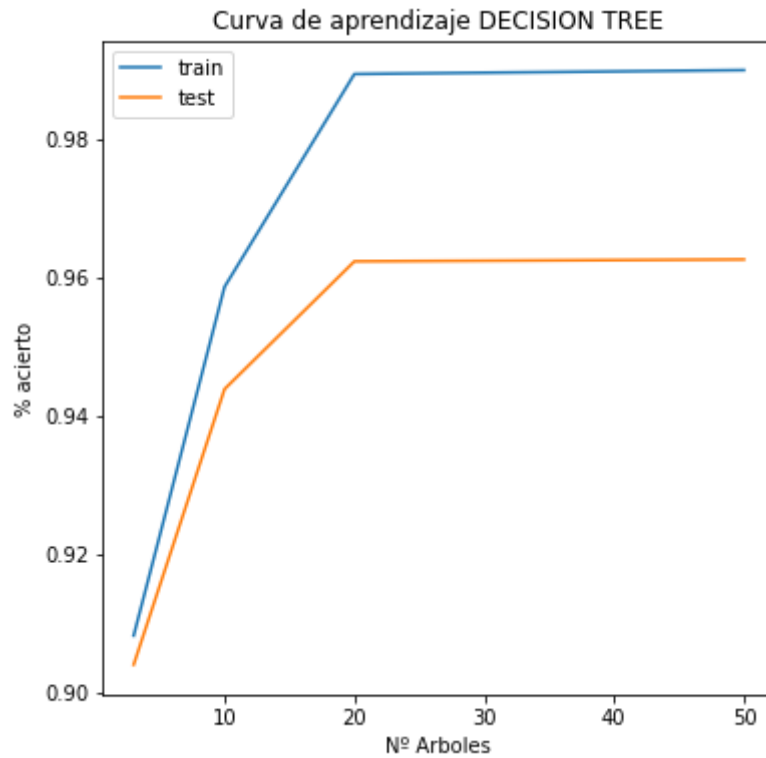


Fig 11. Curva de aprendizaje - DT

3.1.2 RANDOM FOREST

Para este algoritmo se usó el hiper parámetro de número de árboles,se usaron distintas profundidades del árbol (5,10,20,100 y 150 árboles).

En la siguiente imagen se logra ver el código empleado para hallar los mejores hiper parámetros en el algoritmo Random forest:

```

RF (Random Forest)

def experimentar_rf(num_trees, numero_de_variables, X, Y):
    folds = 10
    skf = KFold(n_splits=folds)
    resultados = pd.DataFrame()
    idx = 0
    for trees in num_trees:
        for num_variables in numero_de_variables:
            # para almacenar los errores intermedios
            EficienciaTrain = []
            EficienciaVal = []
            Macc = []
            Mpre = []
            Mrec = []
            Mf1 = []
            for train, test in skf.split(X, Y):
                Xtrain = X[train,:]
                Ytrain = Y[train]
                Xtest = X[test,:]
                Ytest = Y[test]
                #llamada a la función para crear y entrenar el modelo usando los datos de entrenamiento
                modelo = RandomForestClassifier(n_estimators=trees, max_features=num_variables, criterion="gini")
                modelo.fit(Xtrain,Ytrain)
                #predecir muestras de entrenamiento
                Ytrain_pred = modelo.predict(Xtrain)
                #predecir muestras de pruebas
                Ytest = modelo.predict(Xtest)
                #evaluamos las predicciones del modelo con los datos de test
                EficienciaTrain.append(np.mean(Ytrain_pred.ravel() == Ytrain.ravel()))
                EficienciaVal.append(np.mean(Ytest.ravel() == Ytest.ravel()))
                Macc.append(accuracy_score(Ytest, Ytest))
                Mf1.append(f1_score(Ytest, Ytest))

            resultados.loc[idx,'numero de arboles'] = trees
            resultados.loc[idx,'variables para la selección del mejor umbral'] = num_variables
            resultados.loc[idx,'eficiencia de entrenamiento'] = np.mean(EficienciaTrain)
            resultados.loc[idx,'desviacion estandar entrenamiento'] = np.std(EficienciaTrain)
            resultados.loc[idx,'eficiencia de prueba'] = np.mean(EficienciaVal)
            resultados.loc[idx,'Intervalo de confianza (prueba)'] = np.std(EficienciaVal)
            resultados.loc[idx,'accuracy real'] = np.mean(Macc)
            resultados.loc[idx,'f1_score'] = np.mean(Mf1)
            idx= idx +1
        print(f"termina para {trees} arboles")
    return (resultados)

```

Fig 12. Código hiper parámetros RF.

Se puede observar que para el algoritmo Random forest los mejores resultados se obtienen con una cantidad de árboles de 50.

	número de árboles	variables para la selección del mejor umbral	eficiencia de entrenamiento	desviacion estandar entrenamiento	eficiencia de prueba	Intervalo de confianza (prueba)	accuracy real	f1_score
0	5.0	5.0	0.988090	0.000627	0.968154	0.012302	0.968154	0.971618
1	10.0	5.0	0.989236	0.000231	0.971592	0.011267	0.971592	0.974629
2	20.0	5.0	0.989648	0.000376	0.971049	0.010867	0.971049	0.974178
3	50.0	5.0	0.989879	0.000318	0.973040	0.009548	0.973040	0.975936
4	100.0	5.0	0.989889	0.000318	0.972406	0.011275	0.972406	0.975385
5	150.0	5.0	0.989889	0.000318	0.972407	0.010215	0.972407	0.975425

Fig 13. Resultados - RF

Después se realiza la curva de aprendizaje, dando el siguiente resultado:

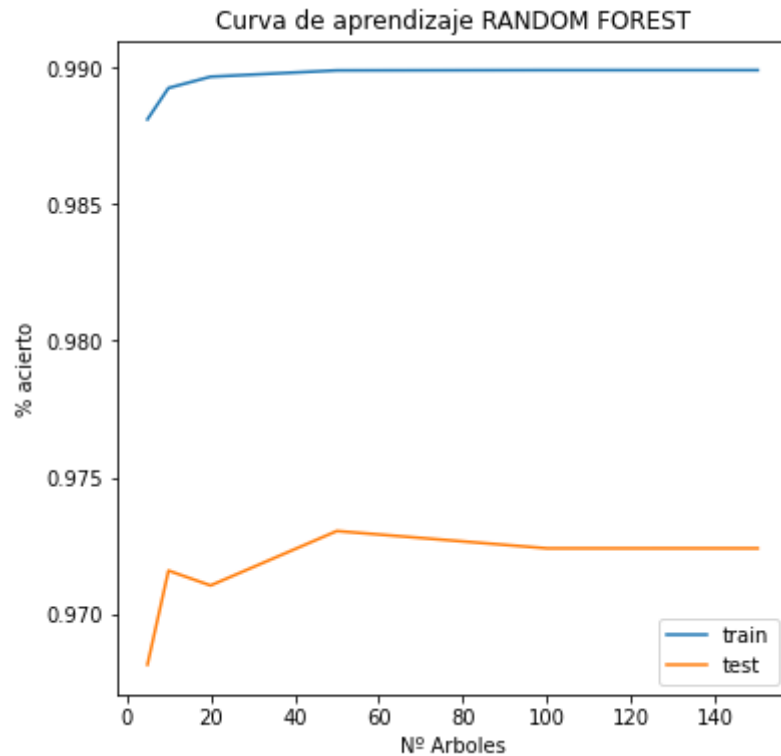


Fig 14. Curva de aprendizaje - RF

3.2 HIPER PARÁMETROS PARA DOS ALGORITMOS PREDICTIVOS + NO SUPERVISADO PCA

Con la finalidad de encontrar los mejores hiper parámetros para los algoritmos predictivos previamente mencionados más el algoritmo no supervisado (PCA), se procedió a iterar en cada fold el hiper parámetro. Además de hacer una transformación PCA. Luego se obtuvo la eficiencia de entrenamiento y de testeo para cada combinación.

3.2.1 DECISION TREE

En la siguiente imagen se puede observar el código empleado para el caso de los algoritmos Decision Tree + PCA:

PCA EN DT

```
def experimentar_dt_pca(num_componentes, depths, X, Y):  
    folds = 4  
    skf = KFold(n_splits=folds)  
    resultados = pd.DataFrame()  
    idx = 0  
    for num_comp in num_componentes:  
        for depth in depths:  
            Mf1 = []  
            EficienciaTrain = []  
            EficienciaVal = []  
            for train, test in skf.split(X, Y):  
                Xtrain = X[train,:]  
                Ytrain = Y[train]  
                Xtest = X[test,:]  
                Ytest = Y[test]  
                pca = PCA(n_components=num_comp)  
                pca.fit(Xtrain)  
                Xtrain = pca.transform(Xtrain)  
                Xtest = pca.transform(Xtest)  
                modelo = DecisionTreeClassifier(max_depth=depth)  
                modelo = modelo.fit(Xtrain, Ytrain)  
                Ytrain_pred = modelo.predict(Xtrain)  
                Yest = modelo.predict(Xtest)  
                EficienciaTrain.append(np.mean(Ytrain_pred.ravel() == Ytrain.ravel()))  
                EficienciaVal.append(np.mean(Yest.ravel() == Ytest.ravel()))  
                Mf1.append(f1_score(Ytest, Yest))  
  
            resultados.loc[idx, 'PCA componentes'] = num_comp  
            resultados.loc[idx, 'eficiencia de entrenamiento'] = np.mean(EficienciaTrain)  
            resultados.loc[idx, 'desviacion estandar entrenamiento'] = np.std(EficienciaTrain)  
            resultados.loc[idx, 'eficiencia de prueba'] = np.mean(EficienciaVal)  
            resultados.loc[idx, 'desviacion estandar prueba'] = np.std(EficienciaVal)  
            resultados.loc[idx, 'f1_score'] = np.mean(Mf1)  
            idx = idx + 1  
  
    return (resultados)
```

Fig 15. Código DT + PCA

DTresultadosPCA = experimentar_dt_PCA([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,17,18,19,20,21,22,23,24,25,26,27,28,29],[20], X, Y)
DTresultadosPCA

	PCA componentes	eficiencia de entrenamiento	desviacion estandar entrenamiento	eficiencia de prueba	desviacion estandar prueba	f1_score
0	1.0	0.855331	0.039347	0.689991	0.129403	0.740445
1	2.0	0.976933	0.006208	0.816633	0.108246	0.836508
2	3.0	0.981969	0.004744	0.845671	0.095369	0.861211
3	4.0	0.989205	0.000727	0.858875	0.099309	0.875227
4	5.0	0.989929	0.000685	0.887731	0.087331	0.903298
5	6.0	0.989688	0.001124	0.896054	0.080457	0.908940
6	7.0	0.989658	0.001014	0.896419	0.073406	0.907298
7	8.0	0.989658	0.001014	0.898680	0.069955	0.910145
8	9.0	0.989899	0.000833	0.901847	0.069923	0.913555
9	10.0	0.989809	0.000793	0.905466	0.064216	0.915551
10	11.0	0.989628	0.001027	0.905375	0.065635	0.915550
11	12.0	0.989628	0.001148	0.904832	0.065195	0.915052
12	13.0	0.989386	0.001109	0.897504	0.071879	0.908304
13	14.0	0.989386	0.001109	0.899946	0.069642	0.910777
14	15.0	0.989085	0.001355	0.899041	0.073181	0.910150
15	17.0	0.989115	0.001331	0.908088	0.062501	0.918321
16	18.0	0.989417	0.001094	0.899313	0.070759	0.911266
17	19.0	0.989628	0.000915	0.899403	0.072526	0.911196
18	20.0	0.989055	0.000867	0.901574	0.070512	0.912904
19	21.0	0.989025	0.000927	0.901845	0.070513	0.913035
20	22.0	0.989115	0.000912	0.905645	0.067217	0.916402
21	23.0	0.989055	0.000884	0.906550	0.065709	0.917174
22	24.0	0.989417	0.000750	0.903564	0.068678	0.915284
23	25.0	0.989417	0.000750	0.905736	0.065797	0.917055
24	26.0	0.989567	0.000872	0.903294	0.067116	0.915289
25	27.0	0.989567	0.000872	0.905374	0.064330	0.916885
26	28.0	0.989567	0.000872	0.907183	0.063574	0.918152
27	29.0	0.989567	0.000872	0.904379	0.065943	0.915910

Fig 16. Resultados DT + PCA.

Se puede observar que en este caso los mejores resultados se obtienen con un número de componentes de 15.

Después se realiza la curva de aprendizaje, dando el siguiente resultado:

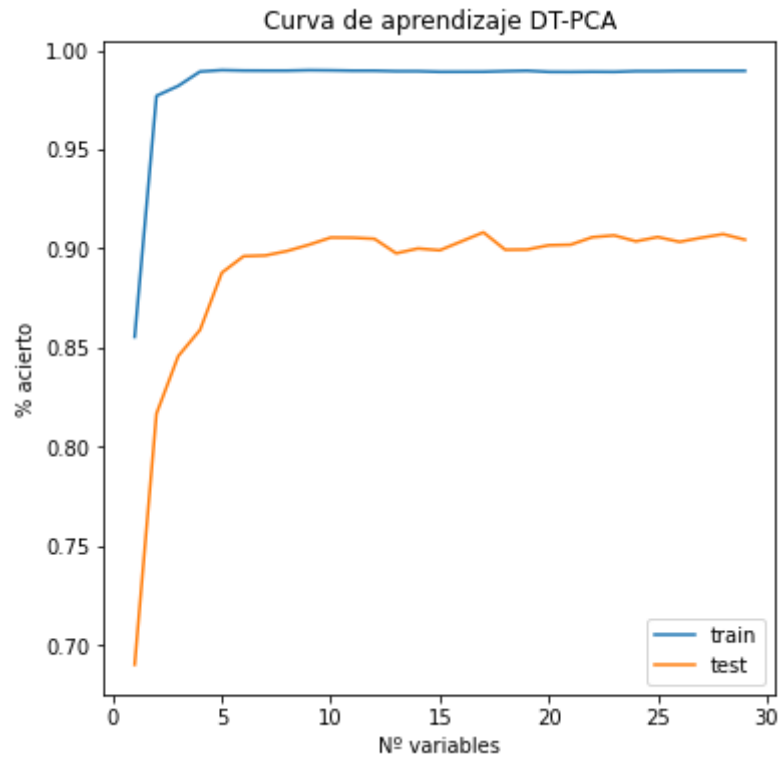


Fig 17. Curva de aprendizaje - DT + PCA

3.2.2 RANDOM FOREST

Código empleado para el caso de los algoritmos Random Forest + PCA:

```
[ ] def experimentar_rf_PCA(num_componentes, X, Y):
    folds = 10
    skf = StratifiedKFold(n_splits=folds)
    resultados = pd.DataFrame()
    idx = 0
    for num_comp in num_componentes:
        EficienciaTrain = []
        EficienciaVal = []
        MFI = []
        for train, test in skf.split(X, Y):
            Xtrain = X[train,:]
            Ytrain = Y[train]
            Xtest = X[test,:]
            Ytest = Y[test]

            pca = PCA(n_components=num_comp)
            pca.fit(Xtrain)
            Xtrain = pca.transform(Ytrain)
            Xtest = pca.transform(Xtest)
            modelo = RandomForestClassifier(n_estimators=50, criterion="gini")
            modelo.fit(Xtrain, Ytrain)
            Ytrain_pred = modelo.predict(Xtrain)
            Ytest = modelo.predict(Xtest)
            EficienciaTrain.append(np.mean(Ytrain_pred.ravel() == Ytrain.ravel()))
            EficienciaVal.append(np.mean(Ytest.ravel() == Ytest.ravel()))
            MFI.append(f1_score(Ytest, Ytest))

        resultados.loc[idx, 'PCA componentes'] = num_comp
        resultados.loc[idx, 'eficiencia de entrenamiento'] = np.mean(EficienciaTrain)
        resultados.loc[idx, 'desviacion estandar entrenamiento'] = np.std(EficienciaTrain)
        resultados.loc[idx, 'eficiencia de prueba'] = np.mean(EficienciaVal)
        resultados.loc[idx, 'desviacion estandar prueba'] = np.std(EficienciaVal)
        resultados.loc[idx, 'f1_score'] = np.mean(MFI)
        idx = idx + 1

    return (resultados)
```

Fig 18. Código RF + PCA

RFresultadosPCA = experimental_rf_PCA([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,17,18,19,20,21,22,23,24,25,26,27,28,29],X, Y)
RFresultadosPCA

	PCA componentes	eficiencia de entrenamiento	desviacion estandar entrenamiento	eficiencia de prueba	desviacion estandar prueba	f1_score
0	1.0	0.987869	0.000312	0.831800	0.098498	0.852786
1	2.0	0.989658	0.000501	0.897939	0.059584	0.910552
2	3.0	0.989809	0.000328	0.919202	0.047399	0.928127
3	4.0	0.989738	0.000349	0.935851	0.036957	0.943209
4	5.0	0.989798	0.000340	0.947070	0.031057	0.953205
5	6.0	0.989829	0.000348	0.953405	0.025271	0.958571
6	7.0	0.989819	0.000345	0.957024	0.021661	0.961685
7	8.0	0.989819	0.000362	0.956391	0.022697	0.961089
8	9.0	0.989829	0.000314	0.957748	0.022064	0.962339
9	10.0	0.989859	0.000335	0.958291	0.021067	0.962795
10	11.0	0.989869	0.000336	0.960282	0.018468	0.964577
11	12.0	0.989859	0.000332	0.961186	0.019062	0.965410
12	13.0	0.989839	0.000313	0.960101	0.019335	0.964529
13	14.0	0.989849	0.000371	0.960644	0.018382	0.964993
14	15.0	0.989819	0.000337	0.961006	0.017753	0.965324
15	17.0	0.989859	0.000347	0.963086	0.018544	0.967088
16	18.0	0.989879	0.000339	0.961911	0.017085	0.966125
17	19.0	0.989879	0.000342	0.961820	0.017292	0.966012
18	20.0	0.989859	0.000341	0.965621	0.014905	0.969359
19	21.0	0.989849	0.000333	0.964715	0.017082	0.968590
20	22.0	0.989889	0.000343	0.965620	0.015317	0.969399
21	23.0	0.989879	0.000330	0.964716	0.016302	0.968596
22	24.0	0.989879	0.000342	0.966887	0.014612	0.970532
23	25.0	0.989859	0.000329	0.965529	0.017844	0.969349
24	26.0	0.989869	0.000330	0.965620	0.016048	0.969448
25	27.0	0.989889	0.000343	0.966435	0.015229	0.970081
26	28.0	0.989879	0.000333	0.966254	0.015205	0.969983
27	29.0	0.989839	0.000349	0.965982	0.015329	0.969693

Fig 19. Resultados RF + PCA.

Se puede observar que en este caso los mejores resultados se obtienen con un número de componentes de 22.

Después se realiza la curva de aprendizaje, dando el siguiente resultado:

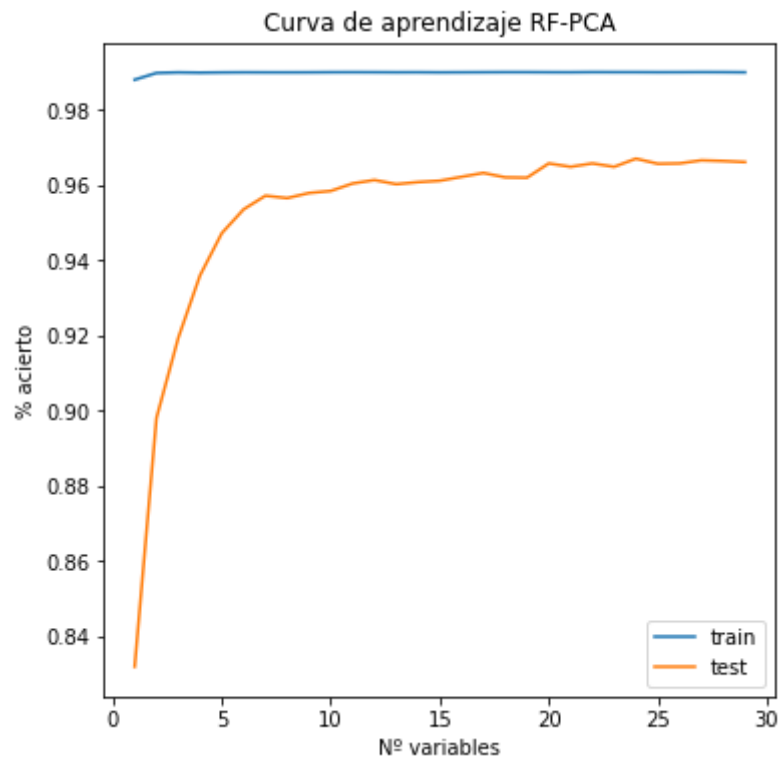


Fig 20. Curva de aprendizaje - RF + PCA

REFERENCIAS

1. UCI Machine Learning Repository: Phishing Websites Data Set. [Online]. Available: <https://archive.ics.uci.edu/ml/datasets/phishing+websites> .