**Trabajo final**

Utilizamos el siguiente dataset de kaggle: <https://www.kaggle.com/eswarchandt/phishing-website-detector#phishing.csv>

El dataset contiene información de mas de 11000 paginas web. De cada pagina web toma 30 parametros y una clase que determina si la web busca realizar phising o no.

Por lo tanto, el objetivo de nuestro trabajo fue el de analizar distitnos modelos que solucionen el problema de clasificacion mencionado anteriormente.

Descargamos el archivo csv y en una notebook de jupyter buscamos distintas soluciones analizando ventajas y desventajas de cada una. Trabajamos con árboles de decision, random forest y el algoritmo KNN. Para cada uno de los tres, variamos las configuraciones de los mismos analizando sus resultados.

Pre-procesamiento

No fue necesario realizar muchas tareas de pre-procesamiento. Luego de leer el csv lo primero que hicimos fue buscar si había alguna variable con algún valor nulo. Como no había ninguna, no tuvimos que utilizar ninguna técnica de imputación de valores.

Por lo tanto, separamos las variables en X (data) e y (target). Luego, descartamos la primera variable de X (index). Esta variable no aporta información alguna. Era simplemente un identificador para cada fila. Dejarla solo implicaría confundir a nuestra solución.

Por ultimo, para separar datos de train y test utilizamos el método hold-out con un 80% de ejemplos para el train.

Arboles de decision

Primero, utilizamos los valores por defecto variando solamente entre gini/entropy (cart/c4.5). Los resultados fueron los siguientes:

|  |  |  |
| --- | --- | --- |
|  | Gini (Cart) | Entropy (C4.5) |
| Train | 99.07271288024427 | 99.07271288024427 |
| Test | 96.11035730438715 | 96.47218453188603 |

Si bien estos resultados son buenos, tiene los valores por defecto min\_samples\_split=2 y min\_samples\_leaf=1 generando así un árbol bastante complejo.

Luego, buscamos la mejor configuración de los árboles de decisión utilizando una validación cruzada de 10 particiones. El resultado fue:

Best params: {'criterion': 'entropy', 'min\_samples\_leaf': 1, 'min\_samples\_split': 2}

Esta configuracion ya la habiamos utilizado al principio. Siendo los resultados:

Train: 99.07271288024427

Test: 96.47218453188603

Este tenia el mismo problema mencionado anteriormente.

El dataset original contiene 30 variables. Nosotros habiamos sacado Index porque no aportaba informacion alguna. Pero ¿existiran otras variables que aporten poca informacion? En caso afirmativo, podemos sacarlas y para tener una solucion mas simple. Esto es posible gracias a la librerira de scikit-learn feature\_selection.SelectFromModel.

En nuestro caso, selecciono 15 variables. Entrenamos el arbol con la configuracion mencionada anteriormente. Los resultados fueron los siguientes:

Train: 98.49598552527424

Test: 95.52238805970148

Como podemos ver, existe una baja en el accuracy. Train: – 0,5767%. Test: - 0,9498%.

Sin embargo, utiliza la mitad de las variables. Dependiendo el caso y el cliente que modelo es preferible. Quizas, de tratarse de un tema vital como puede ser la salud perder casi un 1% en test es un lujo que no pueden darse. Sin embargo, quizas en otros problemas mas triviales es preferible tener un modelo mas sencillo.

De todas maneras, podemos simplificar mas el arbol. Aumentando el valor de min\_samples\_split y min\_samples\_leaf.

Los resultados fueron los siguientes (utilizandos solo las 15 variables seleccionadas):

|  |  |  |
| --- | --- | --- |
| Split/Leaf | Train | Test |
| 2,2 | 97.65916544159222 | 94.61781999095432 |
| 2,4 | 96.77711183987334 | 94.34644957033017 |
| 5,1 | 98.02103358588714 | 94.93441881501583 |
| 5,2 | 97.59131516453692 | 94.5273631840796 |
| 5,4 | 96.79972859889178 | 94.39167797376753 |
| 10,1 | 97.2972972972973 | 94.39167797376753 |
| 10,2 | 97.11636322514984 | 94.39167797376753 |
| 10,4 | 96.70926156281806 | 94.30122116689282 |

Los rendimientos bajan. Dependera del caso para poder decidir si vale la pena o no.

Random Forest

Primero, utilizamos los valores por defecto variando solamente entre gini/entropy (cart/c4.5). Los resultados fueron los siguientes:

|  |  |  |
| --- | --- | --- |
|  | Gini (Cart) | Entropy (C4.5) |
| Train | 99.07271288024427 | 99.07271288024427 |
| Test | 96.8340117593849 | 96.87924016282226 |

Como podemos ver, el rendimiento de test mejora un poco utilizando random forest (0.5% ambos mejores casos), La pregunta vuelve a ser la misma que a lo largo de todo este trabajo. ¿Vale la pena mejorar un 0.5% teniendo una solución mucho mas compleja? Ademas, con random forest perdes la interpretabilidad que te dan los arboles de decisión.

Luego, buscamos su mejor configuracion con gridSearchCV. Resultado:

{'criterion': 'entropy', 'max\_depth': None, 'max\_features': 1, 'min\_samples\_split': 2, 'n\_estimators': 100}

Resultados con esas configuraciones:

Train: 99.07271288024427

Test: 96.96969696969697

Como con los arboles de decision, con el random forest tambien podemos elegir las variables mas importantes. Las 15 variables mas importantes elegidas fueron las mismas seleccionadas con el arbol de decision. Entrenamos el random forest con la configuracion obtenida con GridSearchCV pero solo con las variabels seleccionadas:

Train: 98.49598552527424

Test: 96.33649932157394

Train: – 0,5767%. Test: - 0,6632% con respecto a la configuracion anterior.

KNN

Primero, utilizamos los valores por defecto variando solamente la cantidad de vecinos mas cercanos (K). Resultados:

|  |  |  |
| --- | --- | --- |
| K | Train | Test |
| 1 | 0.9873346149496777 | 0.9615558570782451 |
| 3 | 0.9752346488748163 | 0.9430122116689281 |
| 5 | 0.9646047721361529 | 0.937584803256445 |
| 7 | 0.9590636661766369 | 0.9380370872908186 |

Buscamos la mejor configuracion con GridSearchCV:

{'metric': 'manhattan', 'n\_neighbors': 7, 'weights': 'distance'}

Entrenamos con la mejor configuracion:

Train: 0.9907271288024426

Test: 0.9624604251469923

La desventaja de el algoritmo KNN es que es muy ineficiente en memoria.