UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Aplicación del Aprendizaje Semisupervisado en el descubrimiento de ataques a Sistemas de Recomendación

Presentado por Patricia Hernando Fernández
en Universidad de Burgos — 2 de febrero
de 2023
Tutor: Álvar Arnaiz González

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Álvar Arnaiz González, profesor del departamento de Ingeniería Informática, área de Lenguajes y Sistemas Informáticos.

Expone:

Que la alumna D.ª Patricia Hernando Fernández, con DNI 71362977A, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado «Aplicación del Aprendizaje Semisupervisado en el descubrimiento de ataques a Sistemas de Recomendación».

Y que dicho trabajo ha sido realizado por la alumna bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 2 de febrero de 2023

Vº. Bº. del Tutor:

D. Álvar Arnaiz González

Resumen

En este primer apartado se hace una **breve** presentación del tema que se aborda en el proyecto.

Descriptores

Palabras separadas por comas que identifiquen el contenido del proyecto Ej: servidor web, buscador de vuelos, android ...

Abstract

A **brief** presentation of the topic addressed in the project.

Keywords

keywords separated by commas.

Índice general

Índice de figuras

Índice de tablas

Introducción

1.1. Preámbulo

A diferencia de unas décadas atrás, la sociedad actual está gobernada por los datos. La transición a la era de la información puede ser compleja para determinados colectivos y, consecuentemente, diversos sistemas auxiliares han sido desarrollados con el fin de resumir información y facilitar la toma de decisiones. Entre ellos se encuentran los sistemas de recomendación, que son herramientas que pretenden realizar sugerencias de objetos que pueden resultar interesantes para un determinado perfil.

Económicamente, este tipo de algoritmo es un claro objeto de interés, puesto que puede influir en la toma de decisiones de los compradores y hacer que se inclinen por un determinado producto (por ejemplo, el que tenga una mejor valoración). Los atacantes conocen esta situación y manipulan estas herramientas mediante el uso de perfiles falsos con el fin de beneficiar sus productos o perjudicar los de la competencia.

Este proyecto de investigación pretende explorar cómo el aprendizaje semisupervisado puede ayudar a detectar los ataques a sistemas de recomendación, diferenciando entre perfiles genuinos e inyectados, además de comprobar la veracidad de los planteados por otros investigadores.

Objetivos del proyecto

Este apartado explica de forma precisa y concisa cuales son los objetivos que se persiguen con la realización del proyecto. Se puede distinguir entre los objetivos marcados por los requisitos del software a construir y los objetivos de carácter técnico que plantea a la hora de llevar a la práctica el proyecto.

Conceptos teóricos

Se sintetizarán a continuación algunos de los conceptos teóricos más relevantes para la correcta comprensión del documento.

3.1. Apendizaje automático

Se denomina aprendizaje automático a aquella rama de la inteligencia artificial cuyo objetivo es desarrollar métodos que permitan que un algoritmo mejore su rendimiento mediante la experiencia y procesado de datos. Consecuentemente, los modelos entrenados realizarán predicciones cada vez más precisas como resultado del algoritmo implementado.

Se conoce como «instancia» (o entrada) a cada fila del *dataset*. Estas están caracterizadas por atributos, que pueden ser numéricos (sus valores son números) o categóricos (sus valores son etiquetas, nombres, categorías, etc) [14].

Dentro del aprendizaje automático se diferencian tres grandes grupos en función del tipo de entrada que sea consumida durante el entrenamiento [10]: el aprendizaje supervisado (datos etiquetados), el no supervisado (datos no etiquetados) y el semisupervisado (datos etiquetados y no etiquetados), siendo esta última categoría objeto de estudio en este proyecto de investigación. Se entiende por etiqueta como la representación de una determinada clase.

El aprendizaje supervisado (vertiente predictiva) tiene como meta construir un clasificador (o regresor) que sea capaz de estimar el valor de salida para entradas nuevas (que no se hayan «visto» hasta el momento). Por otro lado, el aprendizaje no supervisado (vertiente descriptiva) trata de inferir la estructura interna de las entradas y agruparlas según similitudes (*clustering*).

Por último, el aprendizaje semisupervisado es la rama que pretende combinar estas dos vertientes complementándolas [10]. La aplicación más común de estos métodos es generar clasificadores efectivos cuando los datos disponibles son escasos, aunque no es la única.

Un clasificador generalmente cuenta con dos fases a lo largo de su ciclo de vida: la fase de entrenamiento y la fase de predicción. En la primera, el estimador trata de encontrar una hipótesis idéntica al concepto objetivo para todos los datos de entrenamiento, suponiendo que la mejor hipótesis para los datos no vistos es aquella que mejor se ajusta a los datos de entrenamiento [30]. Es decir, trata de «aprender» sobre el conjunto de entrenamiento. Posteriormente, el clasificador está preparado para realizar predicciones en instancias nuevas.

El *dataset* disponible inicialmente suele ser dividido en conjuntos más pequeños en función de su finalidad [14]. El conjunto de entrenamiento es aquel subconjunto de los datos utilizado para construir los modelos predictivos (generalmente, a mayor sea este conjunto mejor). Por otro lado, el conjunto de *test* es un subconjunto menor no visto por el modelo (no utilizado durante el entrenamiento) utilizado para estimar el desempeño del modelo a posteriori.

3.2. Aprendizaje semisupervisado

Como se ha mencionado anteriormente, se denomina aprendizaje semisupervisado a aquel conjunto de algoritmos que utiliza datos etiquetados y no etiquetados para realizar tareas de aprendizaje.

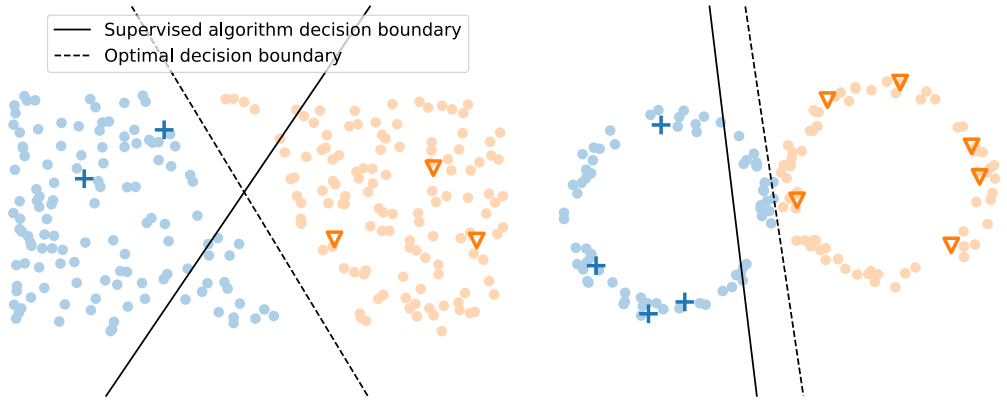
Cuando se habla de aprendizaje semisupervisado, hay ciertas suposiciones que se asumen, además de una condición: que la distribución marginal subyacente en el espacio de entradas $p(x)$ contenga información acerca de la posterior distribución $p(y|x)$ [10]. Si no se cumple este requisito, no se puede segurar que las predicciones basadas en datos no etiquetados vayan a ser adecuadas, aunque en la mayoría de los casos de la vida real ocurre. Sin embargo, la forma en la que $p(x)$ y $p(y|x)$ interactúan no siempre es la misma, aunque se pueden esperar ciertos comportamientos plasmados en las «suposiciones del aprendizaje semisupervisado» (imagen 3.1). Entre ellas:

- ***Smoothness assumption***: si dos muestras x y x' están cerca en el espacio de entrada, sus etiquetas y y y' deberían ser las mismas.

- ***Low-density assumption***: esta suposición propone que la frontera de decisión de un clasificador no debería cortar regiones con alta densidad de datos.

- ***Mainfold assumption***: postula que el espacio de entrada está compuesto por varios *manifolds* («colectores») de dimensiones menores, y que todos los datos contenidos en un mismo *manifold* tienen la misma etiqueta.

Figura 3.1: *Smoothness* y *Low-density assumption* a la izquierda, *Mainfold assumption* a la derecha. Extraída de *A survey on semi-supervised learning* [10].



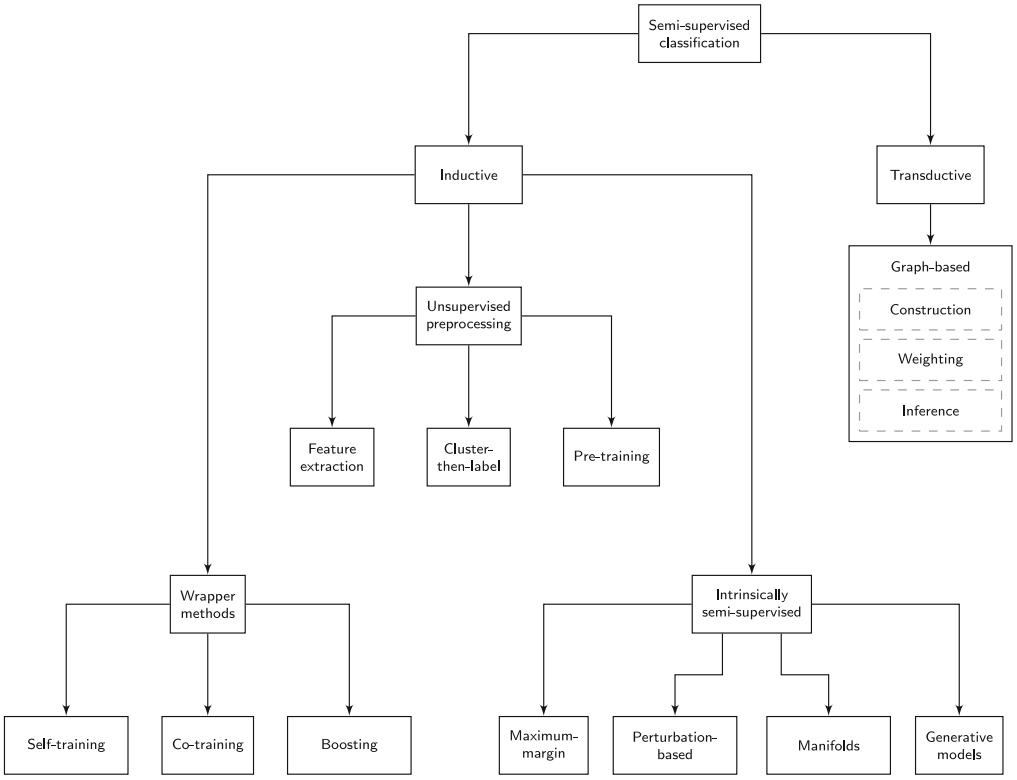
(a) Smoothness and low-density assumptions. **(b)** Manifold assumption.

Clasificación

Inicialmente, se pueden diferenciar dos categorías [10]: los métodos inductivos, cuyo objetivo principal es construir un clasificador que genere predicciones para cualquier entrada y los métodos transductivos, cuyo poder de predicción está limitado a los objetos utilizados en la fase de entrenamiento.

Prescindiendo de los métodos transductivos por ser menos versátiles y útiles en nuestro propósito, los métodos inductivos se subdividen en tres grupos [10]: *wrapper methods* (o métodos de envoltura), *unsupervised preprocessing* y *intrinsically semi-supervised*, siendo materia de estudio los métodos de envoltura.

Figura 3.2: Taxonomía de métodos de aprendizaje semisupervisados. Extraída de *A survey on semi-supervised learning* [10].



Métodos de envoltura

Estos modelos utilizan uno o más clasificadores que son entrenados iterativamente con los datos etiquetados de entrada, además de con datos pseudo-etiquetados. Se denomina pseudo-etiquetado a aquellos datos que inicialmente no estaban etiquetados, pero acabaron estándolo por iteraciones previas de los clasificadores.

Consecuentemente, el procedimiento consta de dos fases que se repiten en cada iteración: el entrenamiento y el pseudo-etiquetado. Durante el entrenamiento, los clasificadores se alimentan de datos etiquetados (o pseudo-etiquetados). En la fase de pseudo-etiquetado, se utilizan datos no etiquetados para que sean procesados por los clasificadores previamente entrenados.

Dentro de esta categoría, se pueden diferenciar tres grandes grupos: *self-training*, que utilizan únicamente un clasificador, *co-training* [10], que utilizan más de uno y los *pseudo-labelled boosting methods*, que construyen

clasificadores individuales que se alimentan de las predicciones más fiables. Se estudiará más en profundidad los métodos *co-training*.

Co-training

En estos algoritmos, varios clasificadores son entrenados iterativamente utilizando datos etiquetados y añadiendo las predicciones (resultados) más confiables al conjunto para ser utilizadas en las siguientes iteraciones [10]. Para que los clasificadores sean capaces de generar información distinta, generalmente se divide el conjunto de entrada según alguna característica (no siendo estrictamente necesario). El *co-forest*, algoritmo protagonista de este proyecto, pertenece a esta categoría.

Para que estos métodos tengan éxito, es importante que los clasificadores base no estén fuertemente correlacionados con sus predicciones, ya que entonces se limita el potencial para proporcionar información útil al resto [10]. Es decir, los estimadores han de ser diversos. Una de las formas de conseguir diversidad es mediante el uso de clasificadores inestables. Otra, obtener más de una vista del conjunto de datos inicial.

3.3. *Ensembles*

Cuando se cuenta con un único estimador base a la hora de realizar una predicción, la etiqueta asignada es directamente la predicha por el clasificador. Sin embargo, en muchas ocasiones, puede resultar enriquecedor recurrir a lo estimado por más de un modelo [32].

Se define *ensemble* como un conjunto de modelos de *machine learning* donde cada estimador base genera una predicción individual que se combina con el resto para generar una salida única [28]. Hay varios motivos por los que resulta favorecedor contar con un conjunto de clasificadores en lugar de con un único estimador base [32]:

- **Motivos estadísticos:** en muchas ocasiones se obtienen buenos resultados en los datos de entrenamiento, pero un error de generalización alto. También se puede dar el caso en el que un conjunto de clasificadores con resultados similares en el conjunto de prueba actúe de manera distinta en la aplicación real (el conjunto de *test* no es lo suficientemente representativo). En estos casos, el hecho de contar con un conjunto de clasificadores reduce considerablemente las probabilidades de predecir mal una etiqueta. Una analogía en la vida real sería cuando

se pide opinión médica a un conjunto de doctores (y no a uno solo) para obtener un diagnóstico no sesgado por las experiencias previas de un sólo médico.

- **Gran cantidad de datos:** cuando la cantidad de información es demasiada, puede no ser manejada correctamente por un único clasificador. Por ello, es mejor particionarla en distintos conjuntos de entrenamiento y prueba, y alimentar cada modelo con una partición.

- **Escasa cantidad de datos:** si los datos no son suficientes, puede ocurrir que un clasificador no sea capaz de aprender su estructura interna. Por ello, es recomendable realizar un muestreo de los datos (con reemplazo) y entrenar varios modelos distintos con ellos. De esta manera, los errores disminuyen.

- **Divide y vencerás:** cuando un problema es demasiado complicado de resolver utilizando un único clasificador, utilizar un conjunto de ellos puede ser efectivo.

Diversidad y clasificación

En la realidad, los datos contienen ruido, *outliers*, distribuciones que se solapan, etc. Por ello, es muy complicado obtener un clasificador con un desempeño perfecto [32]. Los *ensembles* reducen el error cometido por cada estimador base debido a que combinan las salidas de los mismos. Puede parecer «negativo» que los clasificadores individuales difieran. Sin embargo, la realidad es que cuanto más único sea cada estimador utilizado y más distintas sean sus *decision boundaries* (frontera que divide la superficie del problema en partes), mejor.

Esta diversidad se puede obtener de distintas formas, por ejemplo, utilizando distintos conjuntos de entrenamiento y *test* en los estimadores base (utilizando técnicas de *bootstrapping*, por ejemplo). Otra opción muy destacable es utilizando clasificadores «inestables» (aquellos que sean muy distintos con pequeños cambios en el conjunto de entrenamiento), como por ejemplo los árboles.

A la hora de crear un *ensemble*, hay dos preguntas clave que realizarse: ¿cómo se van a generar los clasificadores base? ¿cómo van a diferenciarse los unos de los otros? [32].

Para lograr un mayor nivel de diversidad, se pueden utilizar procedimientos de muestreo o selección de parámetros. En función de ellos y de cómo se

realice el entrenamiento (consultar imagen 3.3), los *ensembles* pueden ser de distintos tipos.

Bagging (Random Forest)

El *bagging* es uno de los modelos más antiguos y populares [32]. Se compone de un conjunto de estimadores base entrenados en paralelo. Por lo tanto, el resultado es el promedio (o más popular) de las salidas de los modelos simples. Para entrenar cada estimador base se utiliza *bootstrapping* [39], que es una técnica de muestreo que genera un subconjunto de datos seleccionando aleatoriamente muestras de un conjunto mayor permitiendo la repetición. Es decir, los clasificadores son entrenados con un subconjunto aleatorio del total de los datos etiquetados como se ilustra en la figura 3.3. Esta técnica puede ser utilizada con o sin reemplazo, entendiendo que el muestreo con reemplazo se produce cuando cada una de las unidades maestrales seleccionada es devuelta a la población total antes de extraer la siguiente (puede repetirse).

El *bagging* es muy útil cuando se dispone poca cantidad de datos, y la cantidad de muestras que contiene cada conjunto de entrenamiento es de entre el $75\,\% - 100\,\%$ del total del *dataset* [32]. Al utilizar *bootstrapping*, los conjuntos de *test* se suelen solapar, además de poder contener muestras repetidas (reemplazo). Por ello, es recomendable utilizar clasificadores base inestables, como por ejemplo los árboles.

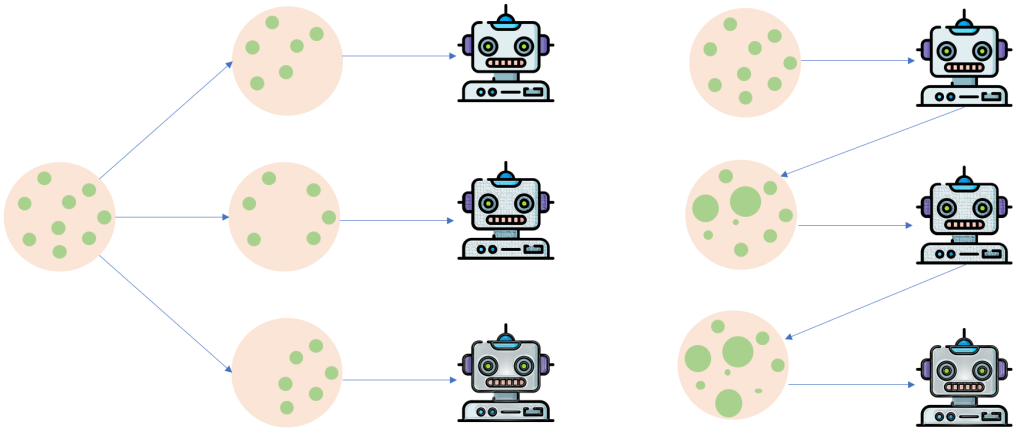
Cuando todos los estimadores base son árboles, se habla de ***random forest***, uno de los métodos de *bagging* más populares que obtiene resultados certeros debido a la aleatoriedad introducida [28]. Cuando el *ensemble* devuelve una etiqueta, es el resultado de una votación realizada por todos los árboles del conjunto. Además de utilizar *bootstrapping*, en el entrenamiento individual de cada árbol, únicamente se «ven» algunos atributos (no el total) para introducir mayor aleatoriedad. Es decir, en cada nodo del árbol solo se tienen en cuenta ciertas características seleccionadas aleatoriamente.

Boosting

Si el anterior modelo utiliza los clasificadores base «en paralelo», este lo haría «en serie» [39]. Es decir, hay un orden secuencial: los estimadores dependen del resultado anterior y tratan de compensar el error que se haya podido cometer (los *weak learners* se convierten en *strong learners* [32]).

El entrenamiento se focaliza, por lo tanto, en torno a las instancias que hayan fallado los clasificadores previos. Esto se consigue dando más peso a

Figura 3.3: Diferencias entre el entrenamiento en *bagging* (izquierda) y *boosting* (derecha).



aquellas muestras mal clasificadas. Por ejemplo, en problemas de regresión, las predicciones con un mayor error cuadrático medio tendrán más peso para el siguiente modelo. En clasificación, los estimadores se entrenan con aquellas muestras en las que los otros elementos del *ensemble* difieran.

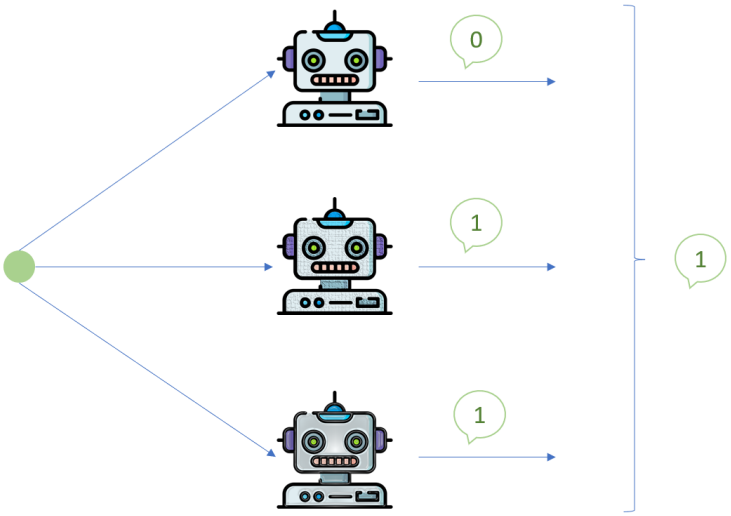
Combinación de los clasificadores

Suponiendo que sólo se dispone de las etiquetas asignadas por los clasificadores base como salida, hay diversos modos de «fusionar» los *outputs* en uno único [32].

En el caso de que todos los estimadores base sean «iguales», es frecuente utilizar *majority voting* («votación de la mayoría»). Cuenta con tres versiones: la votación unánime (todos los clasificadores coinciden), la mayoría simple (coincidencia de más del 50 % de los votos) y la votación por mayoría (la etiqueta que más votos haya recibido, representado en la ilustración 3.4).

Sin embargo, si se sabe que algunos clasificadores son mejores que otros, se puede aplicar *weighted majority voting*, donde las etiquetas predichas por los estimadores con mejor rendimiento tienen más peso que el resto (se asigna una proporción).

Figura 3.4: Representación de la combinación mediante votación.



3.4. Métricas de evaluación

En este trabajo de investigación se van a realizar continuamente comparaciones entre modelos de clasificación. Para ello, es necesario definir unas métricas. Estas se basan en conceptos que se exponen a continuación [14]:

- **Verdadero positivo o *True positive***: se obtiene cuando un clasificador acierta en su predicción y esta es la clase de interés. Generalmente es la clase «positiva» y, en el caso de la aplicación en la detección de ataques a sistemas de recomendación, la «minoritaria».

- **Verdadero negativo o *True negative***: resultado obtenido cuando un clasificador acierta en su predicción y esta no es la clase de interés.

- **Falso positivo o *False positive***: se obtiene cuando un clasificador falla en su predicción y predice que es la clase de interés, cuando en realidad no lo es.

- **Falso negativo o *False negative***: resultado obtenido cuando un clasificador falla en su predicción y afirma que una muestra no pertenece a la clase de interés, cuando en realidad sí pertenece.

- **Total**: la suma de los cuatro puntos anteriores equivale al total de predicciones ($T = TP + TN + FP + FN$).

- ***Class imbalance problem***: es un problema que se genera cuando hay muy pocas instancias positivas en el conjunto de datos en comparación con las negativas (es decir, cuando hay un desequilibro evidente entre ambas clases). Generalmente, causa que la *accuracy* (consultar ecuación 3.1) sea muy elevada, pero no se detecten instancias de la clase positiva, generando modelos inservibles.

Dadas las definiciones anteriores, se facilitan las siguientes métricas utilizadas [14]:

- ***Accuracy*, exactitud o porcentaje de acierto**: mide la tasa de aciertos de un determinado clasificador. A lo largo del proyecto, también es llamada «*score*».

$$Accuracy = \frac{|Aciertos|}{|Total|} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

- ***Error rate*, o porcentaje de error**: mide la tasa de error del clasificador.

$$ErrorRate = \frac{|Fallos|}{|Total|} = \frac{FP + FN}{TP + TN + FP + FN} = 1 - Accuracy \tag{3.2}$$

- **Precisión**: mide cuántas predicciones son verdaderamente positivas entre todas las seleccionadas como tal.

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

- ***Recall*, *sensitivity* o TPR (*true positive rate*)**: determina la capacidad del clasificador para detectar clases positivas. En otras palabras, determina el porcentaje de positivos detectados sobre el total de positivos reales.

$$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

- ***FPR (false positives rate)***: determina cuántos negativos son clasificados como positivos sobre el total de negativos.

$$FPR = \frac{FP}{FP + TN} \tag{3.5}$$

- ***Specificity***: determina cuántos elementos negativos son detectados.

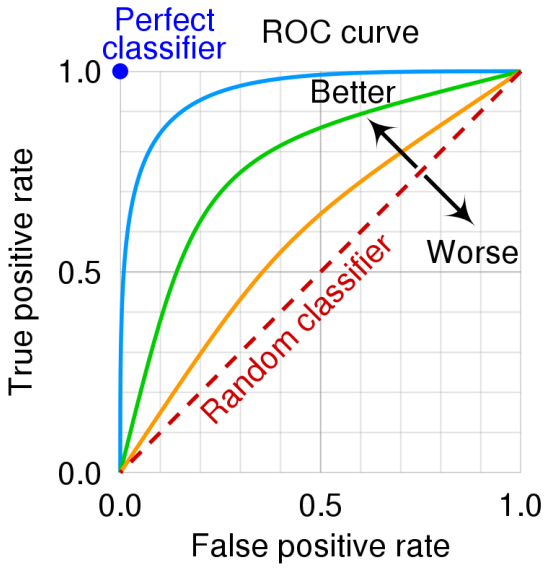
$$Specificity = \frac{TN}{TN + FP} \tag{3.6}$$

- ***F-Measure***: medida que combina la *precision* y el *recall*. Es de interés puesto que la meta general suele ser maximizar estas dos medidas.

$$\text{F-Measure} = 2 \times \frac{precision \times recall}{precision + recall} = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{3.7}$$

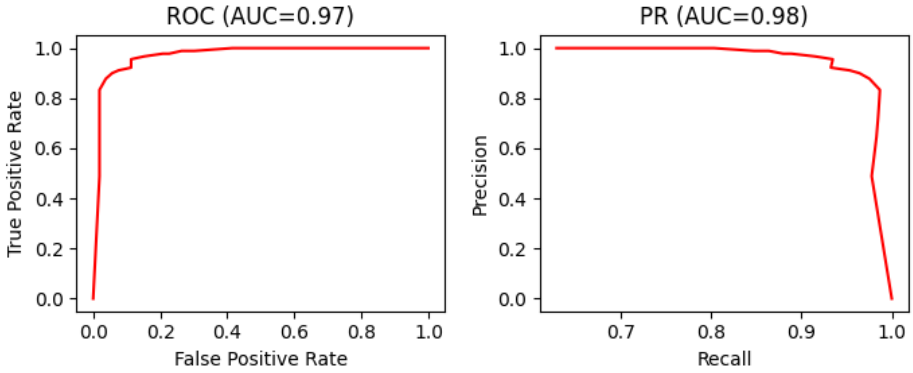
- **Curva *ROC* (*operating characteristic curve*)**: Representación gráfica del *recall* frente a la *specificity* [8]. Un clasificador perfecto tendría una «curva» equivalente a un punto en la esquina superior izquierda de la gráfica, mientras que el peor clasificador (uno aleatorio) obtendría una recta diagonal. Si la curva está por debajo de esta recta que divide el área en dos, significa que las predicciones están siendo «invertidas». Esta medida es popular debido a que no se ve afectada por problemas de clases no balanceadas [5]. Sin embargo, sí que es sensible cuando hay muy pocas instancias positivas en el *dataset* y las estimaciones pueden no ser certeras [13].

Figura 3.5: Calidad del clasificador en función de su curva *ROC* [1].



- **Curva *PR* (*precision-recall curve*)**: Representación gráfica de la precisión frente al *recall*. Cuando se presenta un *class imbalance problem* con muy pocas clases positivas, es adecuado utilizar una curva de estas características debido a que la *precision* y el *recall* permiten evaluar la actuación del clasificador en la clase minoritaria [19].
- ***AUC* (*area under curve*)**: Número decimal que representa el área bajo una determinada curva. Cuando una curva es perfecta (ya sea *ROC* o *PR*), su área es 1.

Figura 3.6: Representación de las curvas *ROC* y *PR* junto con el *AUC* obtenido por cada una para un *co-forest* entrenado con un *dataset* binario.



3.5. *Co-forest*

Descripción

El denominado *co-forest* [28] es un método de clasificación para aprendizaje semisupervisado (más concretamente, de envoltura) que permite construir un *ensemble* de árboles de decisión (clasificación). Este método está basado en el *random forest* y se podría considerar su «versión semisupervisada».

Que sea un algoritmo semisupervisado implica que, durante la fase de entrenamiento, además de utilizar el conjunto de datos etiquetados (L), se utilicen también aquellas pseudo-etiquetas de las predicciones con mejor confianza del conjunto de datos no etiquetados (U).

El proceso de entrenamiento se realiza iterativamente [39], y en cada una de estas repeticiones, se examina cada árbol y se entrena individualmente con un subconjunto de L (L_i) y con un subconjunto de U ($L'_{i,t}$) formado por aquellas pseudo-etiquetas que, además de pertenecer a la submuestra

(aleatoria), posean un nivel de confianza superior a un umbral definido por el usuario (θ). Quienes estiman esta confianza de predicción para las muestras seleccionadas son el conjunto de todos los árboles que forman el *ensemble* menos el árbol que está siendo entrenado individualmente (de ahora en adelante *concomitant ensemble* o conjunto concomitante).

El criterio de parada de la fase de entrenamiento del *co-forest* es el *out-of-bag error* [45] (de ahora en adelante, OOBE). Se define OOBE como el error que comete el conjunto concomitante de un árbol para el total de los datos etiquetados. Es decir, el número de árboles que aciertan la etiqueta entre el número de árboles que votan teniendo en cuenta que, para calcular el OOBE del total de los datos etiquetados, sólo votan aquellos árboles que no hayan utilizado la muestra concreta de L para su entrenamiento.

Algoritmo

Variables utilizadas

Para facilitar la comprensión del pseudocódigo, se definen a continuación algunas de las variables utilizadas:

- n: número total de árboles del ensemble.
- h_i: árbol i-ésimo del conjunto.
- H_i: *concomitant ensemble* o conjunto concomitante del árbol i-ésimo (todos los árboles del *co-forest* menos él).
- x_j: muestra j-ésima (generalmente sin etiqueta).
- $H_i(x_j)$: etiqueta asignada por H_i a la muestra j-ésima.
- $w_{i,t,j}$: confianza de predicción (individual) calculada para una muestra x_j por H_i en la iteración t. Se define confianza como el grado de acuerdo que existe en una votación.
- W: sumatorio de las confianzas de predicción individuales de un conjunto $W = \sum w_{i,t,j}$.
- L: conjunto de datos etiquetados utilizados durante el entrenamiento.
- L_i: subconjunto obtenido tras aplicar *bootstrapping* a L que se utiliza para entrenar el árbol h_i.

- U: conjunto de datos no etiquetados utilizados durante el entrenamiento.
- $U'_{i,t}$: subconjunto aleatorio de U cuya W es menor que W_max (consultar ecuación 3.12).
- $L'_{i,t}$: subconjunto formado por aquellas muestras de $U'_{i,t}$ cuya confianza de predicción es superior a θ.
- θ: nivel de confianza mínimo que tiene que tener el clasificador al predecir la etiqueta de una muestra de U para ser utilizada durante el entrenamiento de un árbol.
- $\hat{e}_{i,t}$: OOBE cometido por H_i en L en el instante de tiempo t.

Pseudocódigo

Se facilita a continuación (pseudocódigo 1) la versión del *co-forest* de la tesis de Engelen [39], basada en la original [28], pero introduciendo los cambios observables en la sección «Discusión de los parámetros del algoritmo».

Fases

En primer lugar, se ha de construir un *random forest* de n árboles. Para introducir aleatoriedad, cada uno de esos árboles es entrenado utilizando un subconjunto aleatorio de L. Es decir, un subconjunto obtenido tras aplicar *bootstrap* a L.

Otro parámetro a tener en cuenta es el número de atributos a considerar en cada árbol de decisión. Por defecto, se ha establecido este valor al log_2 del total (heurística de Breiman [39]). Sin embargo, también podría ser el la raíz cuadrada del total o el total, entre otros.

La segunda fase es entrenar el *random forest* de manera semisupervisada e iterativa hasta que se cumpla el criterio de parada (como se ha mencionado anteriormente, basado en el OOBE). Para ello, se calcula el OOBE de un árbol para una iteración. Si es superior al anterior, se considera que el rendimiento ha empeorado para ese árbol. El algoritmo finaliza cuando todos los árboles empeoran su rendimiento en una determinada iteración.

Por el contrario, en caso de que se haya mejorado, se toma una submuestra de U para pseudo-etiquetar (evidentemente, distinta para cada árbol). Posteriormente se examina cada una de las muestras que forman $U'_{i,t}$, y en

Algoritmo 1: *Co-Forest*

Input: Conjunto de datos etiquetados $L\{(x_i, y_i)\}_{i=1}^{l}$, conjunto de datos no etiquetados $U\{x_i\}_{i=l+1}^{k}$, número de árboles n y umbral de confianza θ

Output: *Ensemble* de árboles entrenado H.

for i = *0, ...,* $n-1$ **do**
  $L_i \leftarrow$ *Bootstrap*(L)
  $h_i =$ EntrenarArbol(L_i)
  $\hat{e}_{i,t} \leftarrow 0{,}5$
  $W_{i,0} \leftarrow min(\frac{1}{10}|\mathrm{U}|, 100)$
end for
$t \leftarrow 0$
while *Algún árbol reciba pseudo-etiquetas* **do**
  $t \leftarrow t+1$
  for i = *0, ...,* $n-1$ **do**
    $\hat{e}_{i,t} \leftarrow$ OOBE(H_i, L)
    $L'_{i,t} \leftarrow \emptyset$
    if $\hat{e}_{i,t} < \hat{e}_{i,t-1}$ **then**
      $W_{max} = \hat{e}_{i,t-1} W_{i,t-1} / \hat{e}_{i,t}$
      $U'_{i,t} \leftarrow$ Submuestrear(U, H_i, W_{max})
      $W_{i,t} \leftarrow 0$
      foreach $x_j \in U'_{i,t}$ **do**
        if *Confianza*$(H_i, x_j) > \theta$ **then**
          $L'_{i,t} \leftarrow L'_{i,t} \cup x_j, H_i(x_j)$
          $W_{i,t} \leftarrow W_{i,t} + Confianza(H_i, x_j)$
        end if
      end foreach
    end if
  end for
  for i = *0, ...,* $n-1$ **do**
    if $(e_{i,t} * W_{i,t} < e_{i,t-1} * W_{i,t-1})$ **then**
      $h_i =$ EntrenarArbol$(L_i \cup L'_{i,t})$
    end if
  end for
end while
return H

caso de que el nivel de confianza supere el umbral, se selecciona esa muestra para el entrenamiento (pasa a formar parte de $L'_{i,t}$).

El último paso sería reentrenar los árboles que hayan cambiado con su propio conjunto inicial de datos etiquetados unido a las pseudo-etiquetas generadas en la correspondiente iteración ($L_i \cup L'_{i,t}$). Es decir, en cada iteración se considera U al completo para poder generar la muestra con la que se pseudo-etiquete, y las anteriores pseudo-etiquetas para un árbol en concreto son descartadas [39].

Tratamiento del ruido y teoría de errores

Como se señala en el *paper* de Zhou y Duan [45], de acuerdo con el trabajo de Angluin y Laird [1], si el tamaño de los datos utilizados en el entrenamiento (m), la tasa de ruido (η), el error de la hipótesis en el peor caso (ϵ) y una constante (c) cumplen la relación de la ecuación 3.8, entonces la hipótesis aprendida por el árbol h_i (que minimiza el desacuerdo en un conjunto de muestras de entrenamiento con ruido) converge a la hipótesis verdadera.

$$m = \frac{c}{\epsilon^2(1-2\eta)^2} \tag{3.8}$$

De acuerdo con [45], se puede obtener la función de utilidad mostrada en la ecuación 3.9 operando en la expresión 3.8.

$$f = \frac{c}{\epsilon^2} = m(1-2\eta)^2 \tag{3.9}$$

Como se ha mostrado en el pseudocódigo, en la iteración i-ésima un determinado árbol se entrena con sus datos etiquetados L_i y un conjunto de pseudo-etiquetas $L'_{i,t}$. Si se considera que el OOBE cometido en L por H_i es $\hat{e}_{i,t}$, entonces se puede estimar que el número de pseudo-etiquetas erróneas en $L'_{i,t}$ equivale a $\hat{e}_{i,t} \times W_{i,t}$ (se recuerda al lector que $W_{i,t}$ es el sumatorio de la confianza de predicción (grado de acuerdo) de H_i en cada muestra de $L'_{i,t}$). Por lo tanto, la tasa de ruido que se encuentra en $L_i \cup L'_{i,t}$ es la estimada por la ecuación 3.10, donde W_0 y η_0 son los parámetros correspondientes a L.

$$\eta = \frac{\eta_0 W_0 + \hat{e}_{i,t} W_{i,t}}{W_0 + W_{i,t}} \tag{3.10}$$

De acuerdo con la ecuación 3.9, la función de utilidad f es inversamente proporcional a ϵ^2. Por lo tanto, si se quiere reducir el error cometido, se debe

aumentar la utilidad de cada árbol en cada iteración [45]. Consecuentemente, se debe cumplir la ecuación 3.11.

$$\frac{\hat{e}_{i,t}}{\hat{e}_{i,t-1}} < \frac{W_{i,t-1}}{W_{i,t}} < 1 \tag{3.11}$$

Discusión acerca de los parámetros del algoritmo

Intuitivamente se puede deducir la ecuación 3.11, ya que el error debe disminuir y la confianza de predicción aumentar con cada iteración. Sin embargo, aunque esto se cumpla, puede ser que se deje de cumplir que $\hat{e}_{i,t}W_{i,t} < \hat{e}_{i,t-1}W_{i,t-1}$, ya que puede ocurrir que $W_{i,t} >>> W_{i,t-1}$. Por este motivo y para cumplir con lo expuesto en la ecuación 3.11, se limita W_{max} como se muestra en la ecuación 3.12 al realizar el muestreo de U en el algoritmo.

$$W_{max} = \frac{\hat{e}_{i,t-1}W_{i,t-1}}{\hat{e}_{i,t}} > W_{i,t} \tag{3.12}$$

El algoritmo original propuesto por [28] y el utilizado en [45] dejan, sin embargo, una cuestión pendiente. Como se puede observar en la ecuación 3.12, W_{max} requiere para calcularse tanto el OOBE como la W obtenida en la iteración anterior, y ambos autores inician W a 0. Esto implica que en la primera iteración $W_{max} = 0$ y, por lo tanto, evita que se realice un muestreo de U para pseudo-etiquetar (pararía el algoritmo). En su tesis, Engelen [39] propone solucionar este problema iniciando $W = min(\frac{1}{10}|U|, 100)$, aunque destaca que imponer esta constante hace que el impacto de los datos sin etiquetar en el algoritmo dependa profundamente del tamaño del *dataset*.

Decisiones de implementación

Además de los aspectos expuestos en el apartado anterior, hay dos decisiones adicionales que se han tomado en la implementación por no estar contempladas en el algoritmo original.

En primer lugar, el valor de W_{max} cuando $\hat{e}_{i,t}$ es 0. Como se puede comprobar en la fórmula 3.12, al estar en el denominador, se produce una indeterminación. Si el valor se sustituye por un número muy cercano a 0, W_{max} tiende a infinito, provocando una submuestra de U ilimitada. Como la función de W_{max} es determinar el tamaño de la submuestra, y en este caso el error del conjunto concomitante es nulo, se ha decidido mantenerlo alto pero limitar a $\theta \times |U|$.

Por otro lado, tampoco se hace explícito cómo iniciar $W_{i,t}$ en aquellas iteraciones en las que no se cumple que $\hat{e}_{i,t} < \hat{e}_{i,t-1}$. Se puede pensar que se debe iniciar a 0. Sin embargo, esto causaría que en iteraciones posteriores nunca se genere una submuestra de U, pues como se puede comprobar en la ecuación 3.12 el tamaño de W_{max} sería 0. Por este motivo, se ha decidido iniciar $W_{i,t} \leftarrow W_{i,t-1}$.

3.6. *Tri-Training*

Descripción

El denominado *tri-training* [46] es un método de clasificación para aprendizaje semisupervisado perteneciente a la rama del *co-training*. Utiliza tres estimadores base (i, j y k) que son entrenados utilizando un conjunto de datos etiquetados L y pseudo-etiquetas generadas por el resto del conjunto.

Dos de las principales características que lo convierten en un algoritmo fácil de utilizar en escenarios reales de minería de datos es que no requiere vistas suficientes y redundantes de los datos ni el uso de distintos algoritmos de aprendizaje supervisado en los clasificadores base [46]. Sin embargo, sí que es un requisito que los estimadores base sean diversos.

El proceso de entrenamiento se realiza iterativamente, actualizando los estimadores base oportunos utilizando L y un subconjunto de U (L_i) pseudo-etiquetado por los dos clasificadores restantes. Este subconjunto se genera (para i) utilizando aquellas instancias de U en las que los estimadores concomitantes de i (es decir, j y k) concuerden en su etiqueta.

Por otro lado, el error para i se aproxima dividiendo el número de muestras de L para las cuales el conjunto concomitante de i realiza una predicción incorrecta entre el número total de muestras de L para las cuales la etiqueta predicha por j es la misma que la predicha por k.

Algoritmo

Variables utilizadas

Para facilitar la comprensión del pseudocódigo, se definen a continuación algunas de las variables utilizadas:

- h_i: estimador i-ésimo del *ensemble* (entrenado con un conjunto concreto de datos que puede variar en cada iteración).

- L: conjunto de datos etiquetados utilizados durante el entrenamiento.
- U: conjunto de datos no etiquetados utilizados durante el entrenamiento.
- S_i: subconjunto obtenido tras realizar *bootstrap* a L para un clasificador en concreto.
- L_i: subconjunto de U pseudo-etiquetado para i.
- l'_i: tamaño del conjunto L_i en la iteración anterior (utilizable para saber si anteriormente un clasificador ha sido entrenado con pseudo-etiquetas o no).
- e_i: error del estimador i.
- e'_i: error anterior del clasificador i.

Pseudocódigo

Se facilita a continuación (pseudocódigo 2) el pseudocódigo original [46] del *tri-training*.

Fases

En primer lugar, se entrenan los estimadores base utilizando un subconjunto aleatorio de L.

La segunda fase es entrenar el *tri-training* de manera semisupervisada e iterativa hasta que los estimadores base no reciban nuevas etiquetas del resto del *ensemble*. Para ello, se comprueba que el error (explicado anteriormente) sea menor que en la iteración anterior, y en ese caso, se genera L_i utilizando aquellas instancias de U en las que los estimadores concomitantes de i (es decir, j y k) concuerden en su etiqueta. En caso de que anteriormente no se haya entrenado con pseudo-etiquetas, se estima el valor que hubiese tenido el conjunto L_i anterior.

Posteriormente, se comprueba si el conjunto L_i nuevo es mayor que el anterior. En caso de que lo sea, se realizan algunas correcciones para evitar violar ecuaciones relacionadas con el tratamiento de errores. Puede llegar a ocurrir que se reduzca el tamaño de L_i durante este proceso.

El último paso sería reentrenar los clasificadores base que hayan recibido nuevas etiquetas con estas mismas unidas a L.

Algoritmo 2: *Tri-Training*

Input: Conjunto de datos etiquetados L, conjunto de datos no etiquetados U, tres clasificadores base.
Output: *Ensemble* de tres estimadores base entrenados.

for i = *1, ..., 3* **do**
  $S_i \leftarrow Bootstrap(L)$
  $h_i \leftarrow$ EntrenarClasificador(S_i)
  $e'_i \leftarrow 0.5$
  $l'_i \leftarrow 0.0$
end for
while *Algún estimador base reciba pseudo-etiquetas* **do**
  for i = *1, ..., 3* **do**
    $L_i \leftarrow \emptyset$
    $actualizar_i \leftarrow$ Falso
    $e_i \leftarrow Error(h_i \& h_k)(j, k \neq i)$
    if $e_i < e'_i$ **then**
      foreach $x \in U$ **do**
        if $h_j(x) = h_k(x)$ $(j, k \neq i)$ **then**
          $L_i \leftarrow L_i \cup \{(x, h_j(x))\}$
        end if
      end foreach
      if $l'_i = 0$ **then**
        $l'_i \leftarrow \lfloor \frac{e_i}{e'_i - e_i} + 1 \rfloor$
      end if
      if $l'_i < |L_i|$ **then**
        if $e_i \times |L_i| < e'_i \times l'_i$ **then**
          $actualizar_i \leftarrow$ Verdadero
        end if
        else if $l'_i > \frac{e_i}{e'_i - e_i}$ **then**
          $|L_i| \leftarrow Submuestrear(L_i, \lceil \frac{e'_i \times l'_i}{e_i} - 1 \rceil)$
          $actualizar_i \leftarrow$ True
        end if
      end if
    end if
  end for
  for i = *1, ...,* 3 **do**
    if $actualizar_i \leftarrow$ *True*
     then
      $h_i =$ Entrenar($L \cup L_i$)
      $e'_i \leftarrow e_i$
      $l'_i \leftarrow |L_i|$
    end if
  end for
end while

Tratamiento del ruido y teoría de errores

En este algoritmo, no es necesario calcular explícitamente la «confianza» del conjunto para cada muestra de U debido a que, para que se añada a L_i, es suficiente con que los otros dos estimadores base del conjunto coincidan en la etiqueta [46]. Puede ocurrir que tanto j como k se equivoquen en una predicción y, por lo tanto, se añada ruido al conjunto de entrenamiento de i. Sin embargo, como se comentó anteriormente en el *co-forest*, de acuerdo con el trabajo de Angluin y Laird [1], si el tamaño de los datos utilizados en el entrenamiento (m), la tasa de ruido (η), el error de la hipótesis en el peor caso (ϵ) y una constante (c) cumplen la relación de la ecuación 3.8, entonces la hipótesis aprendida por el estimador h_i converge a la hipótesis verdadera.

Para cada iteración del algoritmo, se considera un L_i independiente y obtenido considerando todo U. El ratio del ruido de clasificación en la iteración t-ésima se puede calcular mediante la ecuación 3.13, donde $\check{e}_1^t$ es el límite superior del error cometido por j y k en la iteración t, L^t es el conjunto de pseudo-etiquetas para i en la iteración t y η_L es la tasa de ruido en L.

$$\eta^t = \frac{\eta_L|L| + \check{e}_1^t|L^t|}{|L \cup L^t|} \tag{3.13}$$

Como se muestra en la función de utilidad de la ecuación 3.9, como f es inversamente proporcional a $\frac{c}{\epsilon^2}$, entonces se puede deducir que si $f^t > f^{t-1}$ entonces $e^t < e^{t-1}$, lo que implica que el clasificador i va a mejorar si se utilizan las pseudo-etiquetas de L^t en el entrenamiento [46].

3.7. Conceptos básicos de ciberseguridad

Sistemas de cifrado

Un sistema de cifrado es un algoritmo que convierte un texto plano (*input*) en un texto cifrado (*output*). La clave reside en que el nuevo texto es ilegible si se carece de la clave necesaria para descifrarlo, y debe ser lo suficientemente seguro como para que romperlo requiera una cantidad extremadamente grande de recursos (generalmente, tiempo) [3]. El cifrado es aplicable a datos almacenados o «en movimiento» (por ejemplo, intercambio de datos a través de internet).

Cifrado simétrico o criptografía de clave privada

En este tipo de algoritmos, la clave de cifrado y descifrado es la misma (de ahí que deba permanecer «privada»). Se conocen dos tipos, los cifrados de bloque y los cifrados de flujo [25].

En los cifrados de bloque, el texto se divide en bloques de una longitud fija y se cifran bloque a bloque. En función de los modos de operación (los más comunes ECB, CBC, OFM y CTR), también se pueden variar las propiedades de los algoritmos. Algunos de los ejemplos más comunes son DES, Triple DES, AES o *Twofish*.

Cifrado asimétrico o criptografía de clave pública

En este tipo de sistemas, se utiliza una clave para cifrar los datos (la clave pública) y otra distinta para descifrarlos (la clave privada) [20]. Generalmente este tipo de cifrado es mucho más costoso computacionalmente [25], por ello no suelen ser utilizados cuando la cantidad de información a procesar es demasiado grande.

Algunos de los métodos más comunes son RSA o ECC (criptografía de curva elíptica). Dentro de este tipo de algoritmos también se encuentran los sistemas de intercambio de claves, como *Diffie-Hellman*.

Comunicaciones anónimas

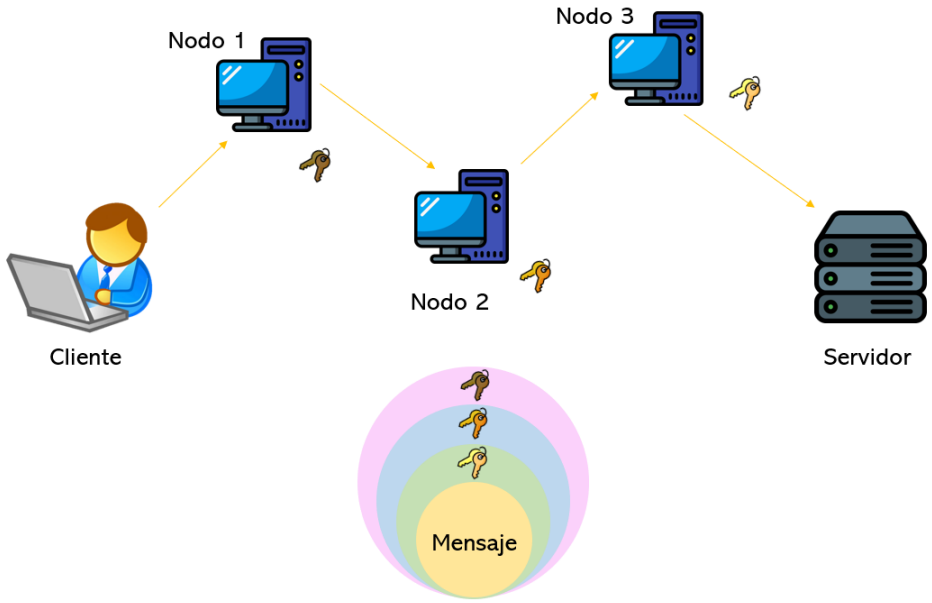
Las redes clásicas son sencillamente rastreables. Póngase un ejemplo: hacer una consulta en internet. En este caso, el ordenador peticionario genera un paquete que circula hasta el *router* de la red a la que esté conectado. Este *router* lo reenvía a los DNS (*domain name service*) del ISP (*internet service provider*) correspondiente, y de ahí puede ir directamente a la página que se desea consultar, o por el contrario pasar por un mayor número de nodos intermedios.

Este camino es relativamente sencillo de rastrear [2], ya que los ISP tienen acceso a los metadatos de las comunicaciones. Adicionalmente, si el contenido de los mensajes se encuentra incorrectamente cifrado, puede derivar en un ataque a los mismos en caso de intercepción por parte de alguno de los nodos intermedios. Se puede pensar que ocultar esta información a terceros es tan sencillo como, por ejemplo, utilizar una de las bien conocidas VPN (*virtual private network*). Sin embargo, los proveedores de servicio de las VPN serán ahora quienes puedan acceder a los metadatos de las comunicaciones [24].

Enrutado cebolla, red Tor y protocolo SOCKS5

Uno de los protocolos que permite ocultar esta información es el llamado enrutado cebolla (*onion routing*) [16], que recibe su nombre por el sistema de capas que utiliza, como se representa en la imagen 3.7. En el mismo, se pueden diferenciar las siguientes fases [24]:

Figura 3.7: Infografía de la red Tor y del enrutado cebolla.



1. Fase uno: la petición se cifra en varias capas (el estándar es 3). Es decir, con la clave pública del nodo de salida, la del nodo intermedio y, por último, al del nodo de entrada. Recordemos que para descifrar una capa, es necesario la clave privada del nodo correspondiente.

2. Fase dos: el servidor de entrada recibe la petición cifrada, y se encarga de descifrar la capa externa (posee la clave), obteniendo la dirección IP del nodo intermedio. De este modo, el servidor puede reenviar el mensaje a este último.

3. Fase tres: el servidor del medio descifra la siguiente capa y encuentra la dirección IP del nodo de salida. Reenvía el resto de la petición cifrada.

4. Fase cuatro: el servidor de salida descifra la última capa y accede a la petición. Sin embargo, no tiene información acerca de quién la realizo. Sí sabe, en su defecto, la dirección del nodo inmediatamente anterior, por lo que puede resolverla y devolvérsela.

La red Tor, utiliza este sistema [2] de enrutado. Para ello, calcula una ruta pseudo-aleatoria hacia destino, obteniendo las claves públicas de los nodos intermedios. Por otro lado, SOCKS5 es un protocolo de Internet usado por Tor [34] que permite redirigir el tráfico a través de la red Tor, actuándo como un proxy de la capa de sesión del protocolo OSI. Por motivos de seguridad, se usará más adelante para ocultar la dirección IP de los investigadores a la hora de extraer vectores de características de sitios *phishing*.

3.8. Ataques a sistemas de recomendación

Los ataques a los sistemas de recomendación (generalmene denominados *shilling attacks* [29] o *profile injection attack* [41]) tienen como objetivo manipular las sugerencias que propone un determinado algoritmo para conseguir que un cliente se incline hacia un elemento deseado. Esta alteración del sistema se consigue inyectando perfiles falsos.

Múltiples estudios se han centrado en formalizar las características de estos ataques con el fin de detectarlos. Entre ellas se encuentran [29]:

- **Intención:** normalmente, se pretende manipular la opinión general acerca de un elemento (ya sea para bien o para mal). Según el objetivo se pueden diferenciar dos tipos de ataques: *push attacks*, que pretenden hacer un objeto más atractivo o *nuke attacks*, cuya intención es la contraria. En caso de que el atacante no busque alterar la opinión acerca de un producto sino restar credibilidad a un sistema (mediante valoraciones aleatorias), se habla de *random vandalism* [6].

- **Fuerza:** la calidad de los ataques se mide teniendo en cuenta el tamaño del relleno (número de valoraciones asignadas a un perfil atacante, que suele rondar entre el 1 y el 20 % del total de los ítems [29]), correspondiente a la ecuación 3.14 y el tamaño del ataque (número de perfiles inyectados en el sistema, rondando entre el 1 y el 15 %), correspondiente a la ecuación 3.15 .

$$Fillersize = \frac{|I_a|}{|I|} \tag{3.14}$$

$$Attacksize = \frac{|U_a|}{|U_g|} \tag{3.15}$$

- **Coste:** se distinguen dos tipos: *knowledge-cost*, que hace referencia al coste de construir perfiles y *deployment-cost*, que es el número de perfiles que se deben inyectar para conseguir un ataque efectivo [41].

Tipos de ataques

En la actualidad se distinguen multitud de ataques distintos. Con el fin de formalizarlos matemáticamente, se han establecido ciertos conjuntos de interés dependiendo de los ítems que contengan [45].

- I_S**:** conjunto de ítems seleccionados para recibir un tratamiento especial (puede ser vacío).
- I_F**:** conjunto de ítems seleccionados para «rellenar».
- I_0**:** conjunto de ítems pertenecientes al sistema de recomendación sin valorar.
- I_t**:** conjunto de ítems objetivo.

Ataques básicos

Se distinguen dos tipos: *random attack* y *average attack* [29]. Ambos tienen parámetros y características muy similares, como se muestra en la tabla 3.1. La principal diferencia reside en que el *average attack* es mucho más potente debido a que cuenta con mayor información acerca del sistema: las valoraciones a los ítems de relleno siguen una distribución $\mathcal{N}(\mu_i, \sigma_i)$, en lugar de $\mathcal{N}(\mu, \sigma)$. Es decir, la valoración para un determinado ítem se adecúa a la distribución concreta de ese ítem en lugar de a la de todo el *dataset*.

Ataques con poco conocimiento del sistema

Los más populares son *bandwagon attack* (o *popular attack*) y *segment attack*. Sus principales rasgos se ilustran en la tabla 3.2.

La principal característica del *bandwagon attack* es que el conjunto I_S ya no está vacío, sino que contiene algunos de los ítems más populares de la base de datos [45]. Estos ítems recibirán también la máxima puntuación posible, de forma que ya no sólo se puntúa el conjunto objetivo. Existe una variante de este ataque llamada *reverse bandwagon attack*, cuyo objetivo es hacer *nuke*, es decir, I_S contiene los ítems menos populares y reciben la puntuación mínima (junto con I_t).

Modelo	$\mathbf{I_S}$	**Valoración** $\mathbf{I_F}$	$\mathbf{I_0}$	**Valoración** $\mathbf{I_t}$
Random	$\emptyset$	Aleatoria siguiendo una distribución normal definida por todas las valoraciones para todos los ítems del sistema $\mathcal{N}(\mu, \sigma)$.	$\emptyset$	máxima o mínima
Average	$\emptyset$	Aleatoria siguiendo una distribución normal definida por las otras valoraciones para ese ítem en concreto $\mathcal{N}(\mu_i, \sigma_i)$.	$\emptyset$	máxima o mínima

Tabla 3.1: Descripción de los ataques básicos. [45]

Modelo	$\mathbf{I_S}$	**Valoración** $\mathbf{I_F}$	$\mathbf{I_0}$	**Valoración** $\mathbf{I_t}$
Bandwagon (average)	Ítems populares (valoración máxima) o ítems desfavorecidos (puntuación mínima) (reverse)	Aleatoria siguiendo una distribución normal definida por las otras valoraciones para ese ítem en concreto $\mathcal{N}(\mu_i, \sigma_i)$.	$\emptyset$	máxima o mínima (reverse)
Bandwagon (random)	Ítems populares (valoración máxima) o ítems desfavorecidos (puntuación mínima) (reverse)	Aleatoria siguiendo una distribución normal definida por todas las valoraciones para todos los ítems del sistema $\mathcal{N}(\mu, \sigma)$.	$\emptyset$	máxima o mínima (reverse)

Tabla 3.2: Descripción de los ataques con poco conocimiento del sistema.

En el *segment attack*, se realiza un pequeño «estudio de mercado» y se introduce en I_S ítems en los que estaría interesado un usuario que fuese a valorar también I_t (de forma que el ataque es más realista).

Ataques con gran conocimiento del sistema

Este tipo de ataques resulta menos relevante que los anteriores debido a la dificultad de su ejecución. En la mayoría de los casos, se necesita una gran

Modelo	**Estrategia de ofuscación**
Noise Injection	$\forall i \in I_F \cup I_S : R_i = r_i + aleatorio * \alpha$
Target Shifting	$\forall i \in I_F \cup I_S : R_i = r_i; I_T : r_{max} - 1$ o $r_{min} + 1$
AOP	I_F escogido del top ítems más populares.

Tabla 3.3: Descripción de las estrategias de ofuscación.

cantidad de información, siendo poco realista que se produzca una situación de estas características en la realidad.

Por ejemplo, el llamado *perfect knowledge attack* [41] basa su efectividad en reproducir la distribución exacta de la base de datos real (exceptuándo los ítems objetivos). El *sampling attack* construye los perfiles a inyectar basándose en una muestra de perfiles reales [29].

Como se puede intuir, conocer datos estadísticos exactos sobre una base de datos o metadatos asociados a perfiles de usuarios es poco realista (cada vez menos debido a las mayores medidas de seguridad) y por lo tanto estos ataques resultan meramente teóricos.

Ataques ofuscados

Los ataques ofuscados [29] se basan en intentar «camuflar» los perfiles inyectados haciéndolos pasar por reales. Algunas de las características de su implementación se pueden consultar en la tabla 3.3.

El ataque de *noise injection* introduce a los conjuntos I_S e I_F un «ruido» (número aleatorio que sigue una distribución Gaussiana) multiplicado por una constante α. *Target shifting* incrementa (o decrementa) en una unidad la valoración de I_t con el fin de crear diferencias entre ataques similares sin influir excesivamente el resultado y el *Average over popular items (AOP)* pretende ofuscar el *average attack* cambiando la forma de selección de I_F (en lugar de seleccionar los ítems del conjunto total de la colección, se seleccionan los X % ítems más populares).

Otros tipos de ataques

Además de los ataques previamente ilustrados, existen otros con objetivos más diversos o estrategias distintas. El anteriormente mencionado *random vandalism* (cuya intención únicamente es degradar la calidad del recomendador para causar descontento entre los usuarios) pertenece a esta categoría. Se pueden distinguir, además, ataques basados en copiar comportamientos de

usuarios influyentes (modelo *PUA* (*Power User Attack*)) o ítems poderosos (modelo *PIA* (*Power Item Attack*)) [29]. Sin embargo, son menos populares.

3.9. *Phishing*

TF-IDF

TF-IDF es un algoritmo que pretende definir la importancia que tiene una determinada palabra dentro de un documento teniendo en cuenta la relación que pueda tener con otros documentos dentro del mismo cuerpo [4]. Es decir, es una especie de «puntuación» para cada palabra del texto que aumenta cada vez que hay una ocurrencia, pero disminuye gradualmente cada vez que aparece en el resto de la colección. Por este motivo, puede ser realmente útil a la hora de extraer *keywords* [44]. Se basa en dos principios:

1. Una palabra que aparezca con mucha frecuencia en un documento tiene especial importancia para el mismo, por lo que es probable que el texto tenga relación con esa palabra.
2. Una palabra que aparezca repetidamente en varios documentos no sirve para identificar uno (o un pequeño subconjunto) en una colección (no sirve como *keyword*), por lo que no es relevante para el mismo.

Para calcular esta puntuación, se han de utilizar las ecuaciones facilitadas a continuación, donde N es el número de documentos disponibles, d es un documento en concreto, D es la colección de todos los documentos y w es una palabra dentro de un documento.

- **TF**: la frecuencia de una palabra w en un documento d se calcula con la ecuación 3.16.

$$\text{tf(w,d)} = \log\left(1 + f(w,d)\right) \tag{3.16}$$

- **IDF**: la frecuencia inversa de una palabra w en la colección d se calcula con la ecuación 3.17, aunque puede implementarse de otras maneras [26] en función del peso que se quiera dar.

$$\text{idf(w,D)} = \log\left(\frac{N}{f(w,D)}\right) \tag{3.17}$$

- **TF-IDF**: la puntuación TF-IDF se obtiene en combinación de las dos puntuaciones anteriores.

$$\text{tf-idf(w,d,D)} = tf(w,d) \times idf(w,D) \tag{3.18}$$

Técnicas y herramientas

4.1. Técnicas

Entornos virtuales? Conda?

Preguntar: las herramientas (nombres propios) en inglés, ¿van en cursiva? cambiar en todo el documento. Ejemplos: windows, python, etc.

4.2. Herramientas

Se muestra a continuación algunas de las herramientas más relevantes en el desarrollo del proyecto.

Librerías

Scikit-Learn

Una de las librerías referentes en el ámbito de *machine learning* y Python. Además de utilizar la vertiente relacionada con aprendizaje automático, también se aprovecharon otras ramas del módulo como la de extracción de características de texto (TF-IDF) [15]. Debido a que proporciona una interfaz estándar para los clasificadores base de muchos algoritmos, posee muy buena documentación y es compatible con otras librerías, ha sido muy utilizada.

sslearn

Librería desarrollada por José Luis Garrido-Labrador dedicada al aprendizaje semisupervisado en Python. Utilizada para realizar comparaciones con algunos de los algoritmos implementados [15].

LAMDA

Toolkit escrito en Python con algunas implementaciones de los algoritmos más relevantes de aprendizaje semisupervisado [22]. Nuevamente, ha sido utilizada como referente a la hora de realizar comparaciones contra las implementaciones propias [23].

Beautiful soup

Biblioteca de Python utilizada para extraer datos de ficheros `HTML` obtenidos mediante la realización de *web scraping* a la hora de sintetizar vectores de características para la detección de *phishing*. Se ha recurrido a ella, principalmente, cuando la obtención de ciertos campos en las etiquetas puede ser vulnerada mediante el uso de expresiones regulares corrientes [36].

NLTK

Toolkit de Python utilizado para trabajar con lenguaje natural. Su principal utilidad ha sido procesar palabras como *tokens* para poder facilitar la implementación de algoritmos más complejos como TD-IDF o para analizar y procesar texto [38].

Otros

Otras de las dependencias (estándar) del proyecto se enumeran a continuación.

- ***Requests***: permite realizar peticiones a distintas URLs, además de especificar parámetros relevantes en las mismas (como *headers*, *cookies*, *proxies* o *timeouts*).
- ***urllib***: procesa URLs y las divide en campos.
- ***re***: utilizada para aplicar expresiones regulares en Python.
- **Numpy, Pandas, Matplotlib**: utilizadas para tratar vectores, operaciones en *arrays*, *dataframes* y representar resultados.

Extensiones y portales

ZenHub

Zenhub es una extensión dedicada a la gestión de proyectos *software* que se integra directamente con Github. Permite visualizar proyectos, *sprints*, gráficos propios de *Scrum* y crear tableros. Debido a que el equipo de desarrollo cuenta con un único integrante, se considera la alternativa óptima (por ejemplo, a Jira) por su sencillez [43].

Github

Portal que permite alojar distintos repositorios en la nube [18]. Ha sido escogido por ser una de las plataformas más populares, además de por permitir la interacción entre distintos usuarios. La gestión del *backlog* del producto ha sido simulada mediante la creación de *issues*.

Programas

KEEL

Herramienta desarrollada en Java por distintas universidades españolas y financiada por el Ministerio de Educación y Ciencia [7]. Proporciona implementaciones de *machine learning*, y ha sido utilizada para probar aquellos algoritmos no disponibles en las librerías de Python mencionadas anteriormente.

Otros

- **TeXStudio**: editor de LaTeX utilizado.
- ***Visual Studio Code***: editor y depurador de código empleado junto a algunas de sus principales extensiones.
- ***Git BASH***: emulador de BASH para Microsoft Windows que proporciona una terminal de línea de comandos de Git.

Scripts

Script para levantar proxies *SOCKS5*

Durante la extracción de vectores de características, se realizan peticiones a páginas de *phishing*. Para garantizar que estas páginas no puedan

rastrear desde donde se ha realizado la petición, se han utilizado *proxies* que implementan el protocolo SOCKS5.

Para ello, se ha implementado en python un *script* auxiliar que levanta en paralelo tantas instancias de *Tor* como se soliciten, y mantiene los hilos vivos hasta que se interrumpa la ejecución del *script*.

Por cada instancia de Tor que se quiera levantar, se necesita un fichero `torrc` en el directorio `/etc/tor/` [33]. Cada una de ellas debe tener, además, su propio puerto de control, su propio puerto *socks* y su directorio de datos. Por ello, se ha creado una clase auxiliar que genera estos ficheros. Para levantar la instancia simplemente ha de ejecutarse el comando `tor -f /etc/tor/torrc.x` (siendo x el número de la instacia correspondiente), aunque se ha decidido, además, dirigir la salida al fichero `/dev/null`. Es importante destacar que el puerto de control debe ser el siguiente al puerto *socks*. Teniendo en cuenta que los puertos por defecto de Tor son el 9050 y el 9051, se puede incrementar partiendo de esos números [47]. Para comprobar que la instancia levantada funciona correctamente, se hace una petición a «http://ipinfo.io/ip» y se comprueba con una expresión regular que la dirección obtenida es la correcta. De este modo, se sabe que el *proxy http* levantado funciona, y se puede utilizar redireccionando las peticiones oportunas a través del *proxy* `socks5h://127.0.0.1:`y (donde y es el puerto *socks* de la instancia correspondiente).

Debido a que el puerto Tor se abre «a la escucha» en la máquina local (está esperando que se realice una conexión), no conlleva ningún riesgo. Para cerrar el puerto, es suficiente con parar el proceso que esté ejecutando el *script*. Se puede comprobar ejecutando en una terminal el comando `sudo lsof -i:`y (nuevamente, y es el puerto *socks* levantado). Cuando el *script* esté funcionando, la salida del comando mostrará diversos campos, como el propio comando (Tor), el *PID*, el nombre (*listen*). Si el *script* se para, el comando no mostrará salida, lo que implica que el puerto no está abierto [17].

Aspectos relevantes del desarrollo del proyecto

5.1. Algoritmos de aprendizaje semisupervisado

A continuación se facilitan los resultados de las experimentaciones realizadas con las implementaciones propias de algunos métodos de aprendizaje semisupervisado.

Para evaluar la calidad de las implementaciones, se han realizado distintos experimentos con algunos de los *dataset* más comunes en métodos de clasificación. En concreto, se han utilizado los que están disponibles en la librería *SKlearn*, que se muestran en la tabla 5.1. El parámetro n representa el número de instancias que contiene un determinado conjunto de datos, mientras que el parámetro m muestra el número de características que contiene cada una de esas instancias.

Co-forest

Para comprobar la correctitud del algoritmo se han realizado diversos experimentos. En primer lugar, se ha evaluado el modelo en diversas situaciones de su ciclo de vida. Posteriormente, se ha realizado una comparativa contra la implementación proporcionada por KEEL.

Experimentación con el algoritmo

Los resultados se pueden observar en las gráficas 5.2, 5.3 y 5.4.

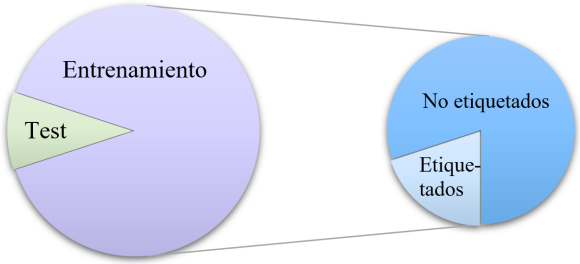
Nombre	**Descripción**	**Clases**	n	m
Iris	Conjunto de instancias pertenecientes a diferentes tipos de plantas de la especie Iris.	3	150	4
Dígitos	Conjunto de instancias que representan una imagen de 8x8 perteneciente a un dígito.	10	1797	64
Vino	Conjunto de instancias pertenecientes a tres clases de vino con sus parámetros estimados mediante análisis químico.	3	178	13
Cáncer de Mama	Conjunto de instancias que representan parámetros de distintas mujeres que pueden padecer o no cáncer (clasificación binaria).	2	569	30

Tabla 5.1: Descripción de los *datasets* utilizados para probar los algoritmos.

- **Fase de entrenamiento:** como se ha desarrollado en los conceptos teóricos, la fase de entrenamiento en el *co-forest* es iterativa, y finaliza cuando ningún árbol recibe nuevas pseudo-etiquetas que puedan cambiar su comportamiento (en la fase de re-entrenamiento).

  Se ha querido estudiar la evolución del *score* (porcentaje de aciertos respecto al total de las predicciones) del algoritmo durante la fase de entrenamiento para los cuatro conjuntos de datos definidos en la tabla 5.1. Para ello, se ha realizado una gráfica ilustrando cómo evoluciona en función de la iteración en la que se encuentre.

Figura 5.1: Gráfica que representa la distribución de los datos.

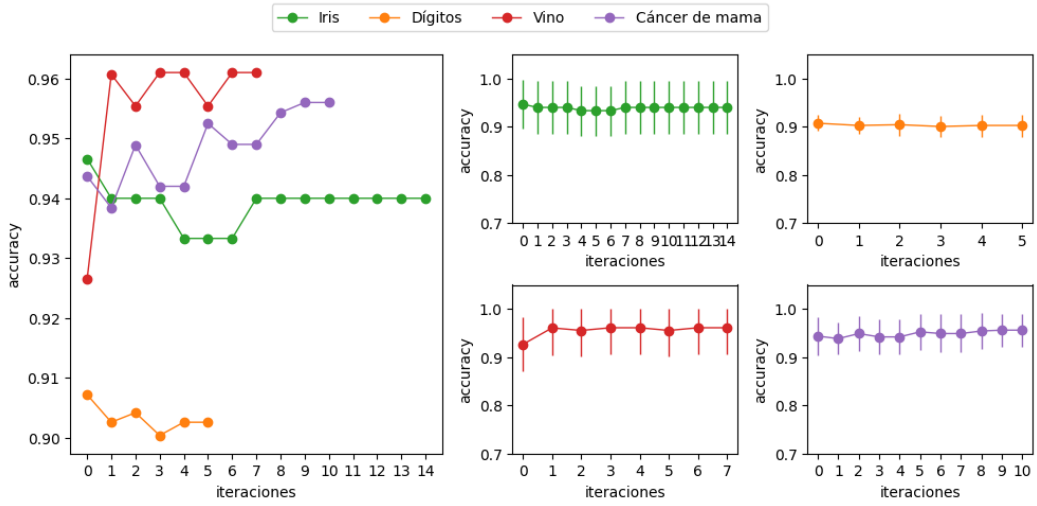


Para garantizar que los resultados obtenidos no son producto de una partición concreta de los datos, se ha realizado validación cruzada (con 10 particiones sin repetición). Por lo tanto, el porcentaje de datos utilizados para el entrenamiento es el 90 % del total (utilizando

estratificación). Los datos de entrenamiento, a su vez, se dividen en etiquetados y no etiquetados. En este caso, el 20 % representa los datos etiquetados, y el 80 % los no etiquetados, como se puede observar en la imagen 5.1. Se han utilizado 20 árboles.

La exactitud media obtenida se puede ver representada en la gráfica 5.2. Es destacable que, dependiendo de los datos que se utilicen para entrenar el *co-forest*, puede variar el número de iteraciones que se necesiten (incluso dentro de un mismo *dataset*). Por ello, siempre se representa el número máximo de iteraciones realizadas, y para no deformar la media, se ha considerado que el valor de las iteraciones inexistentes es el mismo que el valor obtenido en la última iteración (ya que si, el algoritmo siguiese, el resultado devuelto sería igual debido a que no se volvería a entrenar ningún árbol).

Figura 5.2: Gráfica que muestra la evolución de la *accuracy* para cada *dataset* (además de su desviación) durante las iteraciones del entrenamiento del *co-forest*.

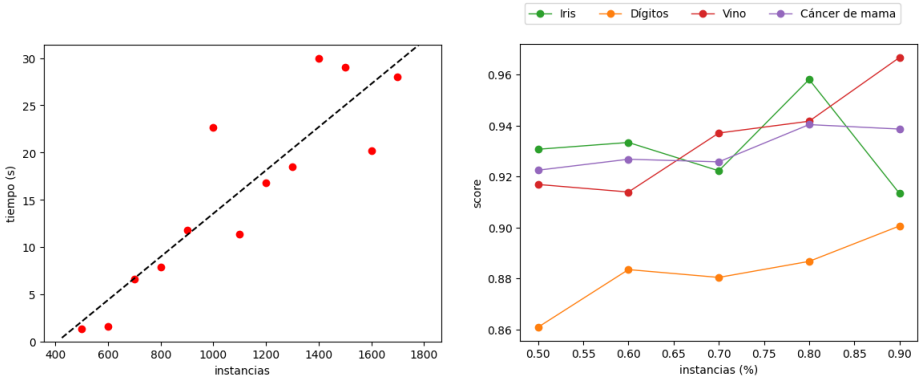


Como se puede comprobar, el modelo mejora los resultados iniciales (exactitud obtenida en la iteración 0, cuando todavía no se ha realizado entrenamiento semi-supervisado y se cuenta con un *random forest* tradicional) en dos de los conjuntos de datos, mientras que en los otros dos empeora. Este comportamiento es lógico debido a que no todos los modelos son apropiados para todos los conjuntos de datos. Si se observa con más detenimiento cada conjunto de datos individualmente en el resto de los cuadrantes de la gráfica 5.2, se puede observar que la

desviación típica varía considerablemente, por lo que se puede deducir que el modelo podría llegar a ser utilizable en la mayoría de los casos si la partición es la adecuada.

- **Tiempo de entrenamiento:** además, también se ha querido evaluar cómo varía el tiempo de entrenamiento en función del número de instancias utilizadas. Para ello, se ha trabajado con el *dataset* que contiene un mayor número de datos («Dígitos»), y los resultados obtenidos se pueden observar en el primer cuadrante de la gráfica 5.3. Como se puede comprobar, sigue un crecimiento aproximadamente lineal. La velocidad, en este caso, es alta, pero cabe recordar que depende en gran medida del número de árboles utilizados y de las iteraciones que estos realicen. Se ha representado en la línea negra punteada, además, el resultado del modelo de regresión lineal.

Figura 5.3: Gráfica que muestra el tiempo de entrenamiento requerido para el *co-forest* en función del número de instancias, además de la evolución del *score* en función del porcentaje total de datos destinado al entrenamiento.



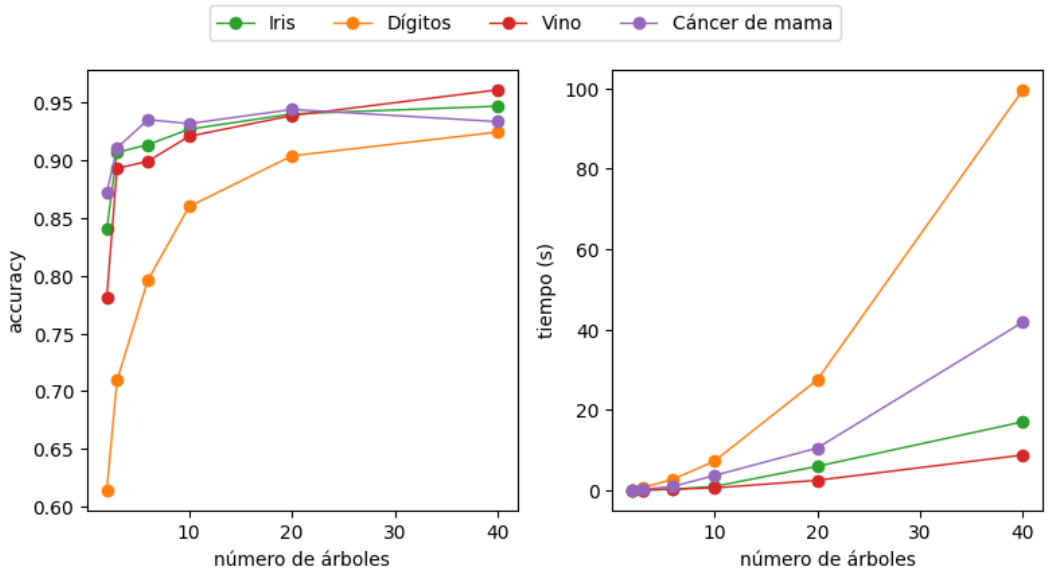
- **Datos de entrenamiento:** En este apartado de la experimentación, se ha querido evaluar cómo se comporta el *co-forest* en función del porcentaje de datos que se utilice para el entrenamiento. Nuevamente, se han utilizado 20 árboles, y la división del conjunto de datos de entrenamiento se ha mantenido: 20 % etiquetados y 80 % no etiquetados. Los conjuntos están estratificados.

  Como se puede comprobar en el segundo cuadrante de la gráfica 5.3, sigue el comportamiento esperado, y por lo general a más instancias utilizadas para entrenar el modelo, mejor desempeño presenta. Nuevamente, recalcar que el resultado obtenido es la media de 10 experimentos realizados.

- **Número de árboles**: Hasta este momento, todos los resultados han sido realizados para $n = 20$. Es decir, utilizando 20 árboles. Sin embargo, el comportamiento del *co-forest* varía considerablemente en función del número de árboles contenidos en el *ensemble*, lo que motiva la realización del siguiente experimento. Nuevamente, se ha desarrollado utilizando cuatro conjuntos de datos y validación cruzada, por lo que se muestra la media en la exactitud para las 10 particiones.

  Como se puede comprobar en la gráfica 5.4, en general, cuantos más árboles se utilicen, mejor *score* alcanza el *co-forest*. Es destacable que, evidentemente, a mayor n, mayor tiempo de procesamiento es requerido, como se ilustra en el segundo cuadrante de esa misma gráfica. Por lo tanto, se debería alcanzar un compromiso. Por lo general, los autores [28] utilizan valores de $n \geq 6$.

Figura 5.4: Gráfica que muestra la exactitud alcanzada por el *co-forest*, además del tiempo de entrenamiento requerido en función del número de árboles utilizados



Comparativa contra KEEL

Para asegurar el correcto funcionamiento del algoritmo implementado, se ha decidido comparar contra la herramienta *KEEL*, creada por distintas universidades españolas y financiada por el Ministerio de Educación y Ciencia.

Parámetro	**Valor**
n	6
θ	0.75
Folds	10
% etiquetados	10 % (en el conjunto de entrenamiento).
Comentarios	Para la comparativa se han utilizado los *datasets* «Iris» y «Vino», ambos estratificados.

Tabla 5.2: Tabla resumen con el diseño del experimento.

Fold	**Iris**		**Vino**	
	Propia	KEEL	Propia	KEEL
1	1.00	0.87	0.89	0.83
2	0.86	0.80	1.00	0.94
3	1.00	1.00	0.94	1.00
4	1.00	1.00	0.94	0.89
5	0.87	1.00	0.65	0.88
6	0.93	0.93	0.94	0.71
7	0.93	1.00	0.83	0.89
8	0.93	0.93	0.94	0.78
9	0.93	0.93	0.67	0.72
10	0.87	0.87	0.94	0.94
Media	0.93	0.93	0.88	0.86

Tabla 5.3: Comparativa entre la *accuracy* del *co-forest* de KEEL y el propio sobre el conjunto de *test*.

En primer lugar y para tener más capacidad de «manipulación», se optó por descargar los ficheros fuentes de la última versión de Github [7] en lugar de utilizar la versión compilada que ofrecen los desarrolladores. Se puede consultar cómo en los anexos.

Para comparar los algoritmos en las condiciones más realistas posibles, se definieron las características mostradas en la tabla 5.2.

Dentro de los ficheros fuente de KEEL, se pueden encontrar distintos conjuntos de datos con las particiones del *K-cross-validation* ya hechas y que son utilizadas en las ejecuciones de sus experimentos. Para comprobar ambos algoritmos en igualdad de condiciones, se convirtieron dichos archivos

`*.dat` en `*.csv` y se importaron en la implementación propia mediante la librería Pandas. De esta manera, el único parámetro que no es idéntico entre ambas implementaciones es qué instancias de entre los datos etiquetados seleccionan los árboles para entrenarse en un primer momento (son aleatorios y generar las pequeñas diferencias observadas).

Los resultados obtenidos se pueden observar en la tabla 5.3. Como se puede comprobar, ambos algoritmos obtienen resultados prácticamente idénticos, siendo un poco superior el porcentaje de acierto logrado por la herramienta implementada propia en el caso del *dataset* «Vino».

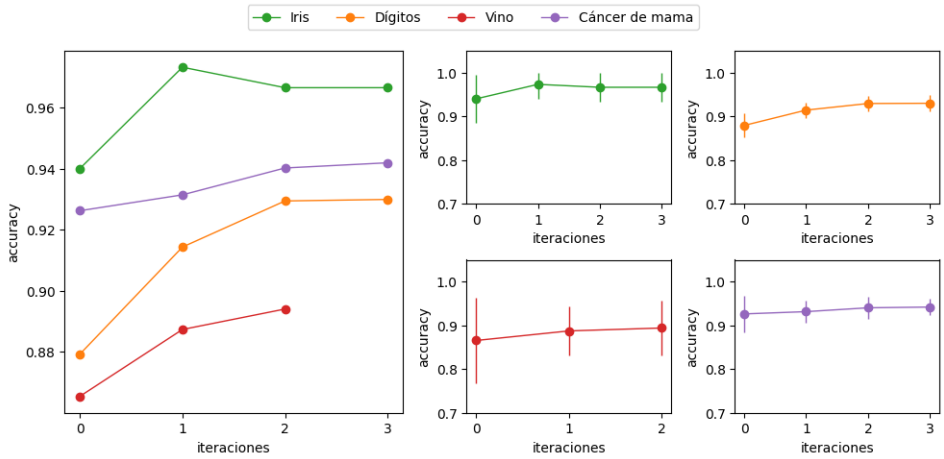
Tri-Training

Al igual que en los algoritmos anteriores, se ha decidido experimentar con distintas opciones a la hora de representar las gráficas, además de realizar una comparativa contra *sslearn* y LAMDA para probar la implementación.

Experimentación con el algoritmo

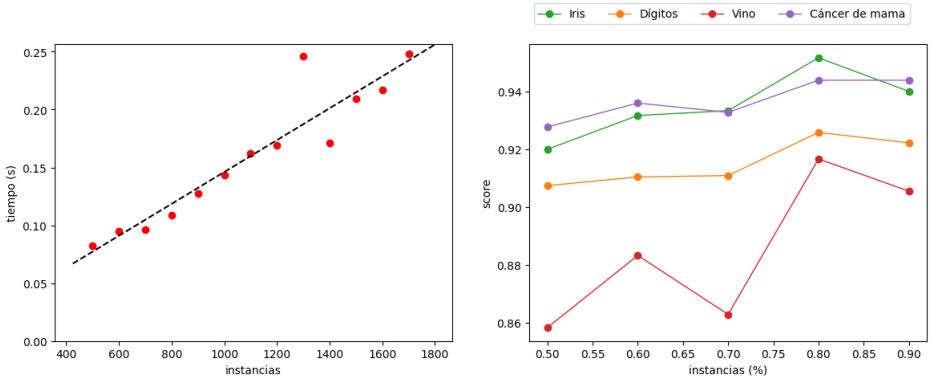
Los resultados se pueden observar en las gráficas 5.5 y 5.6.

Figura 5.5: Gráfica que muestra la evolución de la *accuracy* para cada *dataset* (además de su desviación) durante las iteraciones del entrenamiento del *tri-training*.



- **Fase de entrenamiento:** se ha querido evaluar la evolución de la *accuracy* del modelo en función de la iteración del entrenamiento en la que se encuentre. En este caso, el *tri-training* utiliza como

Figura 5.6: Gráfica que muestra el tiempo de entrenamiento requerido para el *tri-training*, además de la evolución del *score* en función del número de instancias.



estimadores base un *naive-bayes* gaussiano, un árbol de decisión y un *K-neighbors*. Al igual que en el *co-forest*, los resultados representados son la media de 10 experimentos realizados mediante validación cruzada (10 «entrenamientos» distintos) y se pueden observar en la gráfica 5.5, donde el primer cuadrante compara todos los *datasers* y el resto representa la desviación entre las distintas fases de entenamiento para cada *dataset*.

Como se puede comprobar, el modelo mejora los resultados iniciales (exactitud obtenida en la iteración 0, cuando todavía no se ha comenzado el algoritmo de entrenamiento semisupervisado), siendo especialmente notable en los *datasets* «Vino» y «Dígitos». Por ello, se deduce que la incorporación del conjunto de datos no etiquetado durante el entrenamiento es útil y mejora la hipótesis aprendida en cada uno de los clasificadores base (y, por ello, la general del *ensemble*).

- **Tiempo de entrenamiento:** nuevamente se ha trabajado con el *dataset* que contiene un mayor número de datos y los resultados obtenidos se representan en el primer cuadrante de la gráfica 5.6. Como se puede observar, el tiempo crece de forma aproximadamente lineal con una alta velocidad (debido en parte a que este *dataset* no realiza un número elevado de iteraciones durante el entrenamiento).

- **Datos de entrenamiento:** en este caso, se quiere evaluar el desempeño del *tri-training* en función del porcentaje del *dataset* que se utilice para el entrenamiento (posteriormente este conjunto será dividido en L (20 %) y U (80 %)). Los resultados mostrados son la media de 10

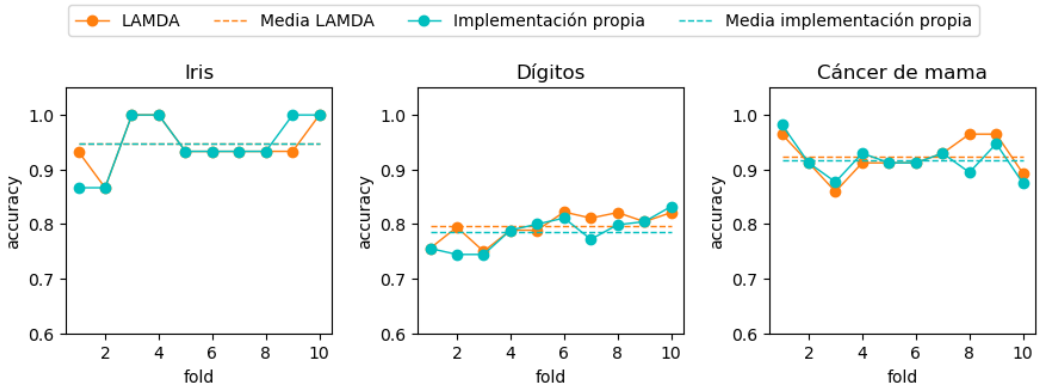
experimentos y se observan en el segundo cuadrante de la gráfica 5.6. En este caso, un buen compromiso sería utilizar un 80 % de los datos, ya que por lo general a mayor número de instancias utilizadas durante el entrenamiento mejor resultado, no alcanzándose siempre la máxima exactitud cuando se usa el mayor porcentaje.

Comparativa contra sslearn y LAMDA

En el caso del *tri-training*, existen implementaciones en `python` que, además, permiten que los estimadores base utilizados sean los disponibles en la librería *Scikit-learn*. Estas son «*sslearn*», biblioteca de aprendizaje semisupervisado escrita por José Luis Garrido-Labrador y disponible en Github [15] y «LAMDA», un *toolkit* [23] desarrollado por Lin-Han Jia (y su equipo) que se encuentra públicamente disponible para su uso [22].

Para la comparativa se ha realizado validación cruzada utilizando 10 *folds*. Evidentemente, ambos modelos han sido entrenados con el mismo conjunto de entrenamiento (más concretamente, idénticos L y U) y probados con el mismo conjunto de *test*. Como estimadores base se ha utilizado el árbol de decisión disponible en la librería *Scikit-learn*.

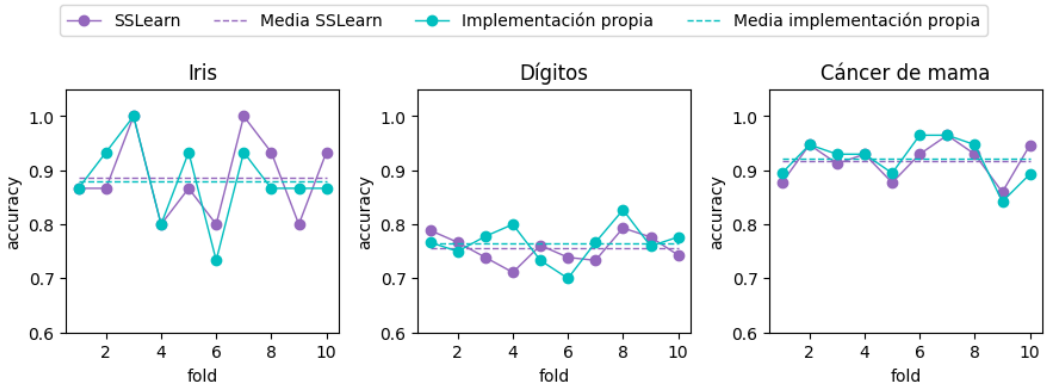
Figura 5.7: Gráfica que representa la variación de la *accuracy* en cada *fold* y la media para la implementación del *tri-training* de LAMDA y la propia.



En el caso de la comparativa contra *sslearn*, el resultado se puede contemplar en la gráfica 5.8, mientras que el caso de LAMDA se observa en la gráfica 5.7. Como se puede comprobar, la media de la *accuracy* para ambos modelos es prácticamente idéntica. Hay algunas variaciones en la *score* para distintos *folds*, pero pueden deberse a detalles de implementación y

selección de los conjuntos internamente. Por este motivo, la implementación se considera correcta.

Figura 5.8: Gráfica que representa la variación de la *accuracy* en cada *fold* y la media para la implementación del *tri-training* de *sslearn* y la propia.



5.2. Detección de ataques en sistemas de recomendación

El objetivo de este apartado en el proyecto es probar nuevos métodos de aprendizaje semisupervisado aplicados a la detección de ataques en sistemas de recomendación. Para lograr este fin, se decidió, en primer lugar, reproducir el artículo original [45], que utiliza el algoritmo *co-forest*. Como los autores no especifican detalles de implementación en su artículo (cómo construir perfiles de ataque, dónde encontrar bases de datos o código, etc.), se concluyó que comparar los resultados obtenidos contra los suyos es la mejor forma de determinar la correctitud del trabajo desarrollado.

En el *paper* original [45] se utilizan tes conjuntos de datos para probar la efectividad del *co-forest*: MovieLens10M, MovieLens25M y Amazon.

MovieLens 10M

Descripción

MovieLens10M [27] es un conjunto de datos utilizado típicamente en la investigación y desarrollo de sistemas de recomendación. Contiene 10 000 054

opiniones generadas por 71 567 usuarios acerca de 10 681 películas. Estas reseñas han sido recopiladas del servicio *online* de películas MovieLens[2].

Los usuarios han sido seleccionados aleatoriamente entre los perfiles que tengan más de 20 películas valoradas y está documentado que todos son auténticos [45] y pertenecen a usuarios reales. Por lo tanto, los perfiles atacantes han de ser construidos mediante programación.

Extracción de vectores de características

Como se puede comprobar en el *paper* de Zhou y Duan [45] los perfiles de usuarios normales son distinguibles de los atacantes debido a su forma de puntuar. En función de estos hábitos, se puede elaborar un método de detección basado en ventanas.

En primer lugar, se dividen todos los ítems del *dataset* en ventanas, consiguiendo un conjunto de J ventanas ($\{w_1, w_2, ..., w_J\}$). No se aplica ningún orden especial, sino que se respeta el que aparece en la propia base de datos. El número de ítems que posee cada ventana se reparte como se muestra en la ecuación 5.2, donde Q es el cociente de dividir el número total de ítems entre J.

$$|w_j| = \begin{cases} Q+1, & j < q \\ Q, & j \geq q \end{cases}$$

Posteriormente se calcula, para cada usuario y ventana, NRW_{u,w_j} y $WNRW_{u,w_j}$ como se muestra en las ecuaciones 5.19 y 5.20 respectivamente. NRW son las siglas de *number of ratings per window*, y $WNRW$ *weighted number of ratings per window*. Es decir, para un determinado usuario, el número de votaciones que haya realizado en una ventana y el ratio de esta respecto al total.

$$NRW_{u,w_j} = \sum_{i \in w_j, r_{u,i} \neq 0} 1 \tag{5.19}$$

$$WNRW_{u,w_j} = \frac{\sum_{i \in w_j, r_{u,i} \neq 0} 1}{\sum_{i \in I, r_{u,i} \neq 0} 1} \tag{5.20}$$

Aplicando estas fórmulas como se muestra en el pseudocódigo 3, se obtienen los vectores de características. Como se puede deducir, este algoritmo

[2]https://movielens.org/

se aplica indistintamente a perfiles genuinos y atacantes, ya que únicamente se precisan las reseñas del usuario. Para una mayor comodidad a la hora de experimentar con estos datos, se han generado ficheros `.csv` con los vectores de características que posteriormente se importarán mediante Pandas para generar conjuntos de entrenamiento y *test*. Debido a que se trata de un problema de clasificación binaria, los perfiles atacantes reciben una etiqueta de 1 y los verdaderos un 0. Cabe destacar, que debido a que en el conjunto de MovieLens todos los usuarios son genuinos, se han de generar las reseñas de los perfiles atacantes.

Algoritmo 3: Algoritmo de generación de vectores de características.

Input: Conjunto de usuarios S, número de ventanas J, conjunto de reseñas del usuario R.
Output: Conjunto de vectores de características X.

$X \leftarrow \emptyset$
for $u \in S$ **do**
  for $j \in J$ **do**
    Calcular NRW_{u,w_j}
    Calcular $WNRW_{u,w_j}$
  end for
  $x_u \leftarrow \{NRW_{u,w_1}, WNRW_{u,w_1}, ..., NRW_{u,w_j}, WNRW_{u,w_j}\}$
  $X \leftarrow X \cup x_u$
end for
return X

Creación de reseñas atacantes

Se ha incluido tres tipos de ataques distintos: *random*, *average* y *bandwagon*. En primer lugar, se han sintetizado las reseñas y posteriormente se han generado los vectores de características idénticamente a los perfiles verdaderos.

La metodología de construcción de las valoraciones sigue los modelos estadísticos expuestos en la sección teórica del proyecto. Sin embargo, se puede visualizar un resumen de los escogidos en la tabla 5.4.

Para construir las reseñas pertenecientes a un ataque del tipo *random*, se ha obtenido, en primer lugar, la media de la puntuación general para todas las películas del sistema y su desviación. Los ítems de relleno han sido seleccionados aleatoriamente entre toda la base de datos (excluyendo,

Modelo	$\mathbf{I_S}$	**Valoración** $\mathbf{I_F}$	**Valoración** $\mathbf{I_t}$
Random	$\emptyset$	Aleatoria siguiendo una distribución $\mathcal{N}(\mu, \sigma)$.	máxima o mínima
Average	$\emptyset$	Aleatoria siguiendo una distribución $\mathcal{N}(\mu_i, \sigma_i)$.	máxima o mínima
Bandwagon (random)	k ítems más populares	Aleatoria siguiendo una distribución $\mathcal{N}(\mu, \sigma)$.	máxima o mínima

Tabla 5.4: Características estadísticas de los tipos de perfiles atacantes.

evidentemente, los ítems objetivo), y se ha asignado a cada uno una puntuación aleatoria siguiendo una distribución normal parametrizada por la media y desviación «global» del sistema. Evidentemente, esta puntuación ha sido corregida para que se encuentre entre los rangos admitidos (por ejemplo, de 0 a 5 estrellas). Debido a que el método de detección utilizado no necesita fechas, el campo perteneciente a la *timestamp* no ha sido rellenado. Posteriormente, se ha asignado la máxima puntuación a los ítems objetivo (*push attack*).

En el caso del ataque *average*, el procedimiento ha sido el mismo, solo que en este caso las puntuaciones de los ítems de relleno son más representativas. En lugar de seguir una normal parametrizada por las características del sistema a nivel global, se ha calculado para cada película su media y su desviación, y se ha asignado una valoración aleatoria que sigue esta distribución. Es decir, cada ítem de relleno recibe una puntuación aleatoria que sigue una distribución normal con media la correspondiente a ese ítem en concreto, y con su respectiva desviación. En caso de que se escoja una película nunca antes valorada, la media utilizada es la media del rango (por ejemplo, 2.5 estrellas) y la desviación es 0.

Por último, el ataque *bandwagon*. Este ataque se realiza exactamente igual que el *random* solo que, además, se añade un conjunto nuevo, los «ítems seleccionados» o I_s. En este caso, se escogen los k ítems más populares de la base de datos (se entiende por más «popular» aquel ítem que posea más valoraciones). Evidentemente, este conjunto se excluye (además de los ítems objetivo) a la hora de escoger los ítems de relleno. La valoración asignada a I_s ha sido o bien la máxima, o bien la mínima (en función de su nota media).

Tipo	**Número**	**Tamaño del relleno**			
		1 %	**3 %**	**5 %**	**10 %**
Genuino	1000	-	-	-	-
Random attack	-	10	10	10	10
Average attack	-	10	10	10	10
Bandwagon attack	-	10	10	10	10

Tabla 5.5: Número y distribución del conjunto de entrenamiento.

Generación de conjuntos de entrenamiento y *test*. Parámetros.

Para generar el conjunto de entrenamiento, se ha utilizado el mismo proceso que el descrito en el *paper* de Zhou y Duan [45]. La distribución de los datos utilizados se muestra en la tabla 5.5.

En primer lugar, se han seleccionado 1000 perfiles aleatorios verdaderos y se han extraído sus vectores de características como se ha indicado previamente. Posteriormente, se han generado reseñas para perfiles de atacantes, y se han extraído sus vectores.

En cuanto al conjunto de test, se han generado 10 conjuntos distintos para garantizar que los resultados de los experimentos no son fruto de una partición concreta de los datos, sino media de una cantidad aceptable. Para ello, se han realizado ficheros `.csv` para distintos porcentajes de tamaño de ataque (1 %, 2 %, 5 % y 10 %) y tamaño de relleno (1 %, 3 %, 5 % y 10 %) de los atacantes. El número de perfiles verdaderos en cada conjunto de *test* equivale a 1000 (y son excluyentes entre ellos y respecto al conjunto de entrenamiento).

En cuanto a los parámetros del algoritmo, se han establecido los mismos que en el *paper* de Zhou y Duan [45]. El tamaño de ventana (J) se ha establecido a 40 y el porcentaje de etiquetas en el conjunto de entrenamiento (d) equivale al 30 %. Como el *paper* no indica qué hacer en el caso del *bandwagon*, se ha decidido que el número de ítems relevantes a evaluar (k) es 30. En cuanto al *co-forest*, el umbral de confianza θ se ha fijado a 0.75 y el número de árboles (n) utilizado es 6.

Cabe destacar que, debido a que es escaso el número de instancias positivas, se ha decidido utilizar como curva para calcular las AUC la curva *precision-recall* (en lugar de la curva ROC).

Resultados

Los resultados se han desglosado en función del tipo de perfiles atacantes inyectados. Se ilustran en las respectivas gráficas: *random attack* 5.9, *average attack* 5.10 y *bandwagon attack* 5.11.

En la leyenda se puede comprobar que se han utilizado distintos algoritmos de ML. Para comparar el *co-forest* con el *random forest*, se han utilizado dos versiones de este mismo (etiquetadas como «RF-A» y «RF-L»). RF-A es un *random forest* que ha sido entrenado con el conjunto de entrenamiento al completo (es decir, con el 100 % de las etiquetas disponibles). RF-L también es un *random forest*, pero la diferencia reside en que, en este caso, únicamente se han utilizado un 30 % de las etiquetas disponibles en su entrenamiento (las mismas que se proporcionan al *co-forest*). Este *ensemble* permite deducir si el *co-forest* (algoritmo semisupervisado) resulta verdaderamente útil en una situación de escasez de etiquetas, o si por el contrario es mejor utilizar la versión supervisada. Por último, también se ha incluído un *tri-training* que utiliza como estimadores base un *naive-bayes* gaussiano, un árbol de decisión y un *K-neighbors*.

Figura 5.9: Gráfica que representa la detección de perfiles que utilizan *random attack* en el *dataset* MovieLens10M.

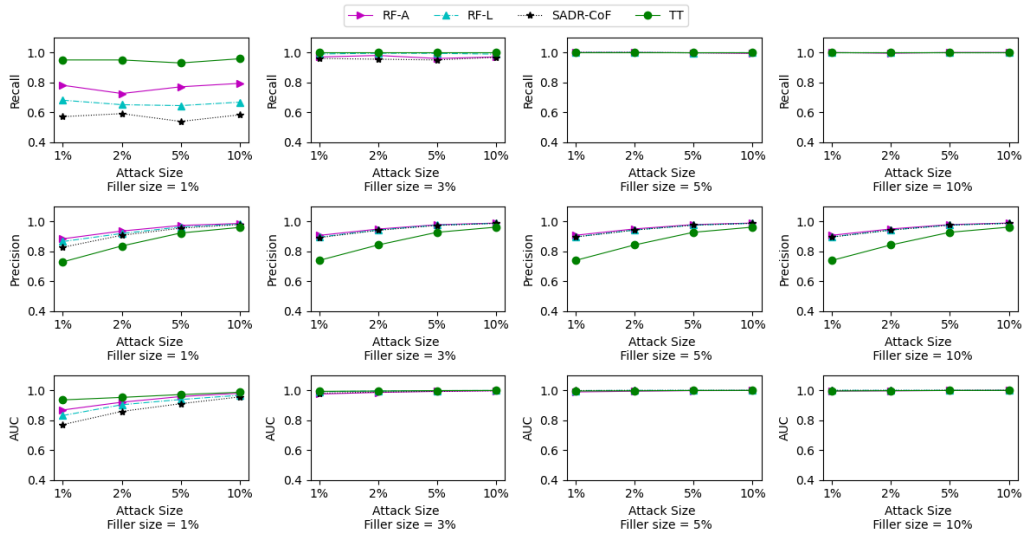
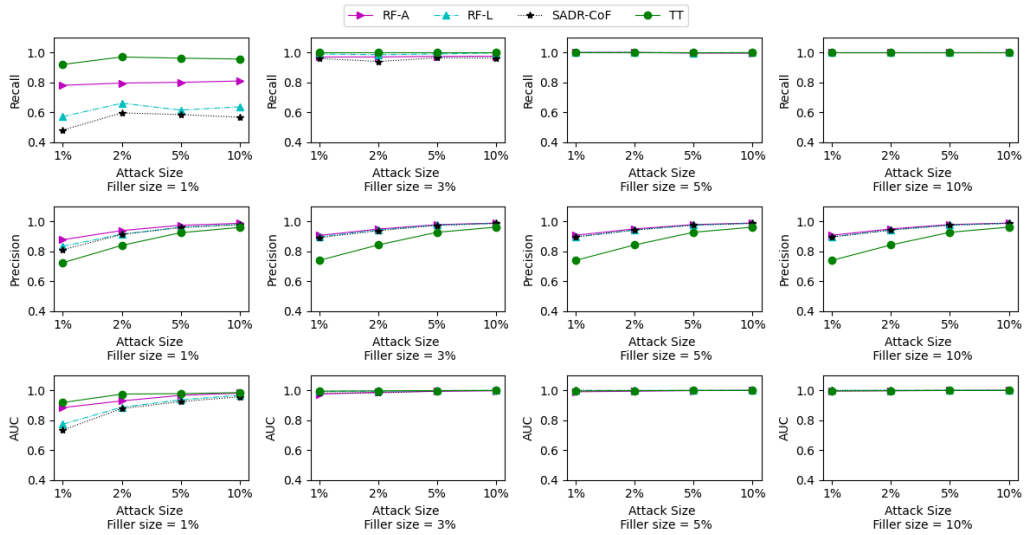


Figura 5.10: Gráfica que representa la detección de perfiles que utilizan *average attack* en el *dataset* MovieLens10M.

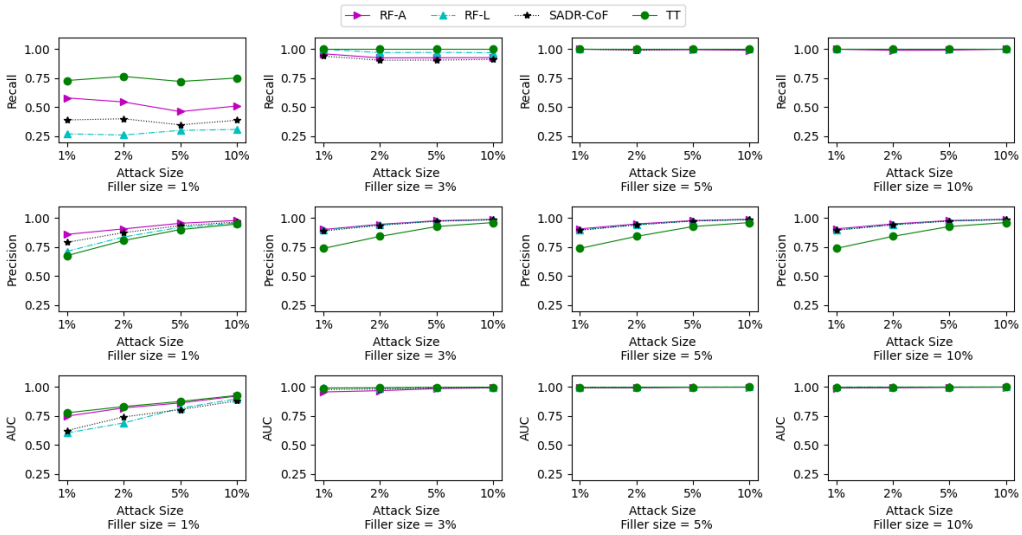


Discusión de los resultados

Puede parecer que los resultados son muy satisfactorios debido a que, en situaciones en las que el *filler size* del atacante es mayor o igual que el 3 %, el *recall* (es decir, la cantidad de instancias positivas que se encuentran) para la mayoría de los algoritmos es muy alto y la precisión (de las instancias que se clasifican como positivas, cuántas lo son realmente) es generalmente buena (exceptuando tal vez, la del *tri-training*, lo que es lógico porque también posee un *recall* mayor).

Sin embargo, una de las fases del ciclo de vida del ML es el preanálisis de datos. Si se observa con detenimiento el *dataset*, se puede comprobar que la media del *filler size* de los usuarios genuinos es aproximadamente de un 1.34 % y que únicamente un 10 % de los usuarios tiene un *filler size* superior al 3 %. Por ello y teniendo en cuenta que para cada conjunto (ya sea de entrenamiento o de *test*) se selecciona aleatoriamente un 1.45 % del total los usuarios genuinos, es muy probable que esté formado por perfiles con un *filler size* inferior al 3 %. Por lo tanto, la única «columna» verdaderamente significativa es la primera, y como se puede comprobar los resultados no son buenos. En otras palabras, es muy probable que el *recall* sea tan bueno en la mayoría de los casos porque los perfiles inyectados tienen un número de valoraciones muy superior a los perfiles genuinos, no porque el método de extracción sea relevante y significativo.

Figura 5.11: Gráfica que representa la detección de perfiles que utilizan *bandwagon attack* en el *dataset* MovieLens10M.



Adicionalmente, se piensa que el método de extracción de vectores de características no es muy adecuado. El hecho de que se aplique directamente en bases de datos sin ningún tipo de ordenación implica que tiene que existir alguna relación subyacente en el orden «por defecto» de los datos y el comportamiento de los usuarios, lo que es una suposición que no puede extrapolarse a todos los conjuntos de datos. Si la base de datos estuviese, por ejemplo, ordenada por popularidad, sería natural que los usuarios votasen más en las primeras ventanas. Si estuviese ordenada por fechas, podría existir una relación entre la edad del usuario y las ventanas en las que vota (películas de su «época»). Sin embargo, al no existir ningún tipo de orden, es una hipótesis demasiado general para ser asumida. Por ello, se ha decidido cerrar esta línea de investigación.

5.3. Detección de *phishing*

Extracción de las URL del *dataset*

Para extraer los vectores de características, en este caso, se necesita extraer el `html` de distintas páginas web (tanto verdaderas como de *phishing*) mediante peticiones de código, por lo que se necesita conocer su dirección. Para garantizar el anonimato en las peticiones que se realizan a páginas de

Dataset	**Instancias**	**Categoría**
Phishtank	1 528	*Phishing*
Openphish	613	*Phishing*
Alexa	1 600	Genuino
Plataforma de pago	66	Genuino
Páginas de banca	50	Genuino

Tabla 5.6: *Dataset* de *phishing*.

phishing, se han utilizado *proxies* con un protocolo *SOCKS5*. Para levantarlos, se ha utilizado un *script* de implementación propia descrito en la sección 4 de este proyecto.

Se ha intentado replicar el *dataset* (procedencia y número) utilizado en el *paper* de Jain y Gupta [21], pero muchos de los enlaces facilitados no están disponibles actualmente. Por ello, se han buscado alternativas. El resultado final se muestra en la tabla 5.6.

En el caso de *Phish Tank*, la base de datos sigue disponible y es accesible en su página web [37]. Debido a que se actualiza a diario, se recupera mediante una petición *get*, aunque se ha tenido dificultades (solucionadas utilizando *proxies* y los *headers* adecuados) debido a que el sitio tiende a bloquear si se hacen unas pocas peticiones seguidas. También sigue accesible la base de *Open Phish* [12], aunque en este caso el número máximo de instancias obtenibles gratuitamente está limitado a 500 y se puede encontrar en un fichero en su página web [11]. Debido a que también se actualiza periódicamente, se recupera en peticiones mediante código.

Los sitios legítimos se han extraído del *top* 1 millón páginas visitadas mediante Alexa, del *top* de plataformas de pago disponibles y de algunos de los sitios de banca más populares. En el caso de los enlaces genuinos, ninguna de las direcciones facilitadas en el *paper* original está disponible. Por ello, los sitios más consultados en Alexa se han obtenido de *Expired domains* [9], las plataformas de pago más comunes de *Shopify* [35] y los sitios bancarios, al igual que en el *paper* original, de *Similar Web* [40] (solo que el número está limitado a 50).

Extracción de vectores de características

...

Trabajos relacionados

Dentro de este proyecto se pueden diferenciar distintas líneas de investigación, principalmente las dirigidas hacia el desarrollo y comprensión de los algoritmos de aprendizaje semisupervisado y las centradas en formalizar ataques a sistemas de recomendación.

Aprendizaje semisupervisado aplicado a la detección de ataques en sistemas de recomendación

Co-Forest aplicado a la detección de ataques [45]

En esta sección, el artículo fundamental es «*Semi-supervised recommendation attack detection based on Co-Forest*» [45]. En este *paper*, se propone un método de detección basado en Co-Forest y se producen distintas comparativas con otros algoritmos para comprobar su eficacia, consiguiendo unos resultados muy aceptables. Partiendo de esta base nace el presente documento, que pretende explorar la solución propuesta por estos autores y expandirla.

Naive Bayes aplicado a la detección de ataques [42]

También es muy relevante citar el trabajo expuesto en «*HySAD: A semi-supervised hybrid shilling attack detector for trustworthy product recommendation*» [42], puesto que propone una aproximación Naive Bayes para separar perfiles de atacantes de perfiles genuinos y además utiliza los tipos de *datasets* que son probados posteriormente por Zhou y Duan (Amazon, Netflix y MovieLens). Se trata de uno de los trabajos de referencia en el área.

Ataques en sistemas de recomendación

La importancia de proteger los sistemas de recomendación ha sido contemplada desde principio de siglo, siendo común la proposición de otros tipos de aprendizaje para detectar los ataques.

Recolección de los tipos de ataques y propuestas de reconocimiento [29]

Respecto a la descripción de los tipos de intrusión, la correcta definición formal (matemática) de sus parámetros y una recopilación de la gran mayoría de ataques existentes, es fundamental referenciar el artículo de Mingdan, Quingshan «*Shilling attacks against collaborative recommender systems: a review*» [29]. Se trata de una recopilación reciente (2018) de las principales investigaciones de los autores más populares en la materia que destaca por su completitud.

Definición de conceptos [41]

Previo a este documento,también es relevante contemplar otros trabajos, como la tesis de William y Mobasher «*Thesis: Profile injection attack detection for securing collaborative recommender systems*» [41]. En ella se introduce el concepto de inyección y se parametrizan características como el tamaño de ataque. Es destacable la autoridad de estos investigadores en la materia, siendo propietarios de muchos documentos de interés.

Primeras definiciones formales [31]

«*Collaborative recommendation: A robustness analysis*» [31], de O'Mahony y Hurley, es uno de los trabajos con más antiguedad pero mayor número de referencias que se encuentra. En él se definen los modelos de construcciones en base a conocimiento del sistema y pone a prueba la robustez de los recomendadores evaluando su estabilidad y precisión ante la presencia de perfiles inyectados (análisis matemático muy completo).

Algoritmos de aprendizaje semisupervisado

Co-Forest

A pesar de que el paper principal del proyecto (Zhou y Duan [45]) expone que se utiliza la versión original del *co-forest* [28], es destacable mencionar que ciertos aspectos de implementación no han sido contemplados en estos

artículos. Por ello, la implementación propia se ha basado también en «*Semi-supervised ensemble learning. Master's thesis*», desarrollada por Engelen y Hoos [39].

Conclusiones y Líneas de trabajo futuras

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

Bibliografía

Dana Angluin and Philip Laird. Learning from noisy examples. *Machine Learning*, 2:343–370, 1988.

Andrea Ardións. Cómo funciona tor y para qué deberíamos utilizarlo, 2017.

Avast. Cifrado de datos: ¿en qué consiste?, 2022.

Marius Borcan. Tf-idf explained and python sklearn implementation, 2020.

PhD. Jason Brownlee. Roc curves and precision-recall curves for imbalanced classification, 2020.

Robin Burke, Michael P. O'Mahony, and Neil J. Hurley. *Robust Collaborative Recommendation*, pages 805–835. Springer US, Boston, MA, 2015.

Jesús Alcalá Fernández (Coordinator). Github: Keel, 2018.

Google Developers. Clasificación: Curva roc y auc, 2022.

Expired Domains. Alexa top websites, 2023.

Jesper Engelen and Holger Hoos. A survey on semi-supervised learning. *Machine Learning*, 109, 02 2020.

Open Fish. Open fish feed file, 2023.

Open Fish. Openphish database, 2023.

Salvador García, Alberto Fernández, Mikel Galar, Ronaldo C. Prati, Bartosz Krawczyk, and Francisco Herrera. *Learning from Imbalanced Data Sets*. Springer International Publishing, 2018.

César García Osorio and Jose Francisco Diez Pastor. *Aprendizaje automático: introducción y problemas tipo.*

José Luis Garrido-Labrador. Github: sslearn, 2023.

Deepan Ghimiray. Avast: qué es tor, es seguro y cómo se usa., 2022.

Vivek Gite. How to check if port is in use on linux or unix, 2023.

Github. Github. let's build from here, 2023.

Haibo He and Yunqian Ma. *Imbalanced Learning: Foundations, Algorithms, and Applications*. Wiley-IEEE Press, 1st edition, 2013.

IBM. ¿qué es el cifrado? definición de cifrado de datos., 2022.

Ankit Kumar Jain and B. B. Gupta. Towards detection of phishing websites on client-side using machine learning based approach. *Telecommunication Systems: Modelling, Analysis, Design and Management*, 68(4):687–700, 2018.

Lin-Han Jia, Lan-Zhe Guo, Zhi Zhou, and Yu-Feng Li. Github: Lamda-ssl, 2022.

Lin-Han Jia, Lan-Zhe Guo, Zhi Zhou, and Yu-Feng Li. Lamda-ssl: Semi-supervised learning in python. *arXiv preprint arXiv:2208.04610*, 2022.

Redacción KeepCoding. ¿qué es el onion routing?, 2022.

Dr. Grzegorz Kołaczek. *Cybersecurity: symetric and asymetric cyphers.*

Scikit learn project. Tf-idf term weighting, 2022.

Group Lens. Movielens datasets.

Ming Li and Zhi-Hua Zhou. Improve computer-aided diagnosis with machine learning techniques using undiagnosed samples. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 37(6):1088–1098, 2007.

Si Mingdan and Qingshan Li. Shilling attacks against collaborative recommender systems: a review. *Artificial Intelligence Review*, 53, 01 2018.

Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

Michael O'Mahony, Neil Hurley, Nicholas Kushmerick, and Guénolé Silvestre. Collaborative recommendation: A robustness analysis. *ACM Trans. Internet Technol.*, 4(4):344–377, nov 2004.

R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006.

Tor Project. I'm supposed to .[ed]it my torrc". what does that mean?, 2022.

Tor Project. Socks5 / socks5, 2022.

Shopify. Payment gateways in spain, 2023.

Beautiful Soup. Beautiful soup documentation, 2023.

Phish Tank. Phish tank database. developer information, 2023.

Natural Language Toolkit. Nltk documentation., 2023.

Jesper Van Engelen and Holger Hoos. Semi-supervised ensemble learning. master's thesis., 07 2018.

Similar Web. Clasificación de los mejores sitios web de créditos y préstamos bancarios en el mundo, 2023.

Chad Williams, Research Advisor, and Bamshad Mobasher. Thesis: Profile injection attack detection for securing collaborative recommender systems, 2006.

Zhiang Wu, Junjie Wu, Jie Cao, and Dacheng Tao. Hysad: A semi-supervised hybrid shilling attack detector for trustworthy product recommendation. page 985–993, 2012.

Zenhub. Zenhub. the productivity platform for software teams, 2023.

Yue Zhang, Jason I. Hong, and Lorrie F. Cranor. Cantina: A content-based approach to detecting phishing web sites. page 639–648, New York, NY, USA, 2007. Association for Computing Machinery.

Quanqiang Zhou and Liangliang Duan. Semi-supervised recommendation attack detection based on co-forest. *Comput. Secur.*, 109(C), oct 2021.

Zhi-Hua Zhou and Ming Li. Tri-training: exploiting unlabeled data using three classifiers. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1529–1541, 2005.

zkilnbqi. How to run multiple tor processes at once with different exit ips?, 2015.