UNIVERSIDAD DE BUENOS AIRES

Facultades de Ciencias Económicas,
Ciencias Exactas y Naturales e Ingeniería

Carrera de Especialización en Seguridad Informática

Trabajo Final

Técnicas de Inteligencia Artificial aplicadas a la SI: Un enfoque actual

Autor:

Lic. Khalil Alejandro Moriello

Tutor:

Dr. Hugo Scolnik

Agosto de 2021 – Cohorte 2020

Declaración Jurada de origen de los contenidos

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

FIRMADO

Khalil Alejandro Moriello

DNI 35375402

Resumen

Hoy en día es sabido que los diseñadores de tecnolog□ras para prevenir ataques informáticos están en constante batalla contra los atacantes. Esto se debe a que las técnicas utilizadas por estos últimos se complejizan y mejoran muchas veces más rápido que las soluciones. Es un aporte interesante estudiar cómo aplicar inteligencia artificial para reducir los tiempos entre el ataque y la detección.

El objetivo de este trabajo final es relevar y comparar el material teórico existente sobre el uso de técnicas de Inteligencia Artificial aplicadas a la Seguridad Informática. Se realizará un estudio del tipo descriptivo: se analizarán los tipos y las características de las distintas soluciones existentes.

Este trabajo servirá como base para la posterior propuesta de mejora sobre algunos de los distintos subtipos y/o técnicas descriptas.

Palabras clave:

- Machine Learning
- Ciberseguridad
- Intrusion Detection System (IDS)
- Detección de malware
- Advanced Persistent Threat (APT)

Índice

Índice de Ilustraciones

Índice de Tablas

Cuerpo introductorio

En el presente trabajo se presentarán dos casos de estudio. El primero está relacionado con la mejora en los sistemas de detección de intrusiones (IDS) y el segundo aplicado al análisis estático de malware.

Para ambos casos se utilizan técnicas de *machine learning* para optimizar los resultados.

Con respecto a los IDS, se parte de los metadatos y se logra una mejora en la detección de patrones que no están especificados en las reglas. Se utilizan muestras reales en crudo, es decir, capturas de tráfico, tanto malicioso como benigno.

Para el análisis estático de malware se muestran las distintas líneas de investigación actuales sin mostrar resultados con pruebas reales.

Como primer punto, se presentan y se describen los distintos algoritmos de *machine learning* utilizados a lo largo del trabajo. El objetivo de dicha mención es brindarle al lector una breve introducción sobre las distintas técnicas y formulación de los algoritmos.

Por último, este trabajo pretende ser la base para la creación de un sistema de detección de malware más avanzado incluyendo el análisis dinámico que queda excluido en esta elaboración.

Capítulo 1: Preliminares

1.1 Introducción

1.1.1 ¿Qué es Inteligencia Artificial?

La inteligencia cubre una gama tan amplia de procesos que es difícil de definir con precisión. Basado en la definición dada por el diccionario, se puede decir que es la capacidad de entender o comprender, o la capacidad de resolver problemas o mismo una habilidad, destreza y experiencia[1].

Tanto los zoólogos como los psicólogos estudian el aprendizaje tanto en animales como en humanos. Existen varios puntos en común entre el aprendizaje animal y el aprendizaje automático. Muchas de estas técnicas derivan de los esfuerzos que los primeros mencionados hicieron para poder describir y estudiar los comportamientos de sus objetos de estudio. [1]

Respecto a las máquinas es posible decir que una máquina aprende cada vez que cambia su estructura, programas o datos, en base a sus entradas o en respuesta a información externa, de manera tal que su desempeño futuro esperado mejora. Un ejemplo de esto último es el reconocimiento de voz que mejora y brinda resultados más precisos después de haber escuchado varias muestras, es decir, que el sistema va aprendiendo en cuanto reciba más entradas, en este caso muestras de habla, y por lo tanto es capaz de dar una salida más precisa. De esta manera, cambia su estructura y así su desempeño futuro mejora.

Alan Turing propuso el test que lleva su nombre que consiste en una prueba que mide la habilidad de una máquina de exhibir inteligencia. Un evaluador humano mantendrá una conversación con un humano y con la máquina en cuestión y deberá identificar cuál es la máquina basándose en la comunicación. Tanto la máquina como el otro humano tratarán de persuadir al evaluador que se está interactuando con otro ser humano. Si el evaluador no es capaz de distinguir a la máquina del humano en la conversación, entonces la máquina es considerada inteligente.

---

[1] https://dle.rae.es/inteligencia

1.1.2 ¿Qué es Machine Learning?

Es la ciencia de conseguir que las computadoras actúen sin haber sido explícitamente programadas. [2] Es una rama de la inteligencia artificial que busca desarrollar técnicas que permitan a las computadoras aprender. [3] Dicho de otra forma, se trata de generar programas capaces de generar comportamiento a partir de información no estructurada suministrada en forma de ejemplos.

Está situada entre las ciencias de la computación y la matemática.

Existen dos formas de clasificar los distintos tipos de aprendizaje para *machine learning:*

- Aprendizaje supervisado: el objetivo principal es aprender un modelo desde un conjunto de entrenamiento que le permitirá al algoritmo hacer predicciones de los datos futuros. Supervisado hace referencia que la salida esperada del conjunto de entrenamiento es ya conocida. Algunos ejemplos son: clasificación, predicción (regresión) y ranking.
- Aprendizaje no-supervisado: el objetivo es ser capaz de reconocer patrones de la entrada del sistema sin contar con un conjunto de entrenamiento. Algunos ejemplos son: *clustering* y asociación.

Algunas definiciones importantes que son necesarias para comprender el vocabulario básico de *Machine Learning*.

- **Overfitting**: sucede cuando el número de parámetros a determinar es mayor o igual que la cantidad de datos. Implica que el algoritmo no generaliza correctamente.
- **Underfitting**: ocurre cuando el modelo no logra capturar la estructura básica de los datos.
- **Variance**: cuántos grados de libertad tiene el resultado de la estimación.
- **Bias**: cuánto se aproxima la estimación al conjunto de entrenamiento.

- **Hiper-parámetros**: son los parámetros que el algoritmo no puede aprender en función de los conjuntos de datos. Deben ser brindados por el usuario.

No Free Lunch theorem (NFL)

"Dos algoritmos de optimización son equivalentes si son promediados para todos los problemas posibles". [4]

Este teorema tiene como implicancia que no existe un algoritmo que sea bueno para cualquier problema, y el hecho de que un algoritmo sea bueno para un problema específico, necesariamente implica que es malo para otro problema.

1.1.3 Técnicas para el uso de los algoritmos de ML

Comprender el problema es el primer y más importante paso en un proyecto de ML. Puede parecer trivial pero la dificultad real y cómo va a ser modelado muchas veces se torna difícil para algunos problemas en particular.

El paso siguiente es conseguir la suficiente cantidad de datos para el problema. Una vez que se obtuvieron las suficientes muestras, es necesario convertirlas para que sea la entrada de los algoritmos. Los datos en crudo no pueden ser usados directamente en los algoritmos de ML ya que tienen ruido, sesgos, valores faltantes y/o incorrectos.

Para los problemas particulares que se estudiarán en este trabajo ya cuentan con los dos pasos anteriores realizados. Como se clasificarán distintos conjuntos de datos, en adelante, *datasets*, se utilizarán algoritmos de aprendizaje supervisado. Para esto, se seguirá una serie de pasos estándares dentro de *machine learning*.

1. Preprocesamiento
2. Aprendizaje
3. Evaluación
4. Predicción

Se utilizarán algoritmos prediseñados dentro de las librerías para *python*[2] *Scikit-learn*[3]. No se pretende hacer una codificación exhaustiva de cada algoritmo, sino más bien a grandes rasgos evaluar la eficiencia de cada uno, sin tener en cuenta optimizaciones de velocidad y eficiencia algorítmica. Para entornos donde es crítica la velocidad y el uso de recursos se recomienda utilizar lenguajes optimizados como C o C++ que ofrecen mejores resultados en comparación con *python*.

Etapa de preprocesamiento

Los datos crudos rara vez vienen con el formato necesario para que la performance de los algoritmos sea óptima. Para ello, antes de aplicar cualquier algoritmo de *machine learning* es necesario hacer preprocesamiento a los datos originales. Este consiste en la utilización de técnicas de reducción de dimensiones, quitar *features* que estén correlacionadas y por lo tanto, redundancia. Otra tarea indispensable es cómo trabajar con las variables categóricas, datos que falten (datos nulos), números fuera de rango. Por último, algunos algoritmos requieren que los datos sean normalizados. [5]

Para el caso particular que se va a tratar en este trabajo, se separarán los datos en dos conjuntos: set de prueba y set de entrenamiento. El set de entrenamiento se utilizará para entrenar los algoritmos y el set de prueba para verificar la *performance* y de esta manera calcular cuán precisa es la clasificación, determinar si hace *overfitting* o *underfitting*.

Etapa de aprendizaje

En esta etapa hay que seleccionar un modelo que mejor ajuste a los datos. Luego viene la validación cruzada, la evaluación de las métricas mencionadas en el apartado anterior y la optimización de hiper-parámetros.

Etapa de evaluación

Una vez que se ha seleccionado un modelo que se ajusta al conjunto de datos de entrenamiento, es posible usar el conjunto de datos de prueba para estimar qué tan bien se desempeña en estos nuevos datos para estimar el error

---

[2] https://www.python.org/

[3] https://scikit-learn.org/

de generalización. Si se está satisfecho con el rendimiento, entonces se puede usar este modelo para predecir nuevos datos.

Es importante tener en cuenta que los parámetros para los procedimientos mencionados anteriormente, como la normalización de *features* y la reducción de dimensiones, se obtienen únicamente del conjunto de datos de entrenamiento, y los mismos parámetros se vuelven a aplicar posteriormente para transformar el conjunto de datos de prueba, así como cualquier nueva muestra de datos. De lo contrario, el rendimiento medido en los datos de prueba puede ser demasiado optimista.

Se definen:

- **Verdadero positivo (TP):** el dato en la muestra es positivo y la prueba predice un positivo.
- **Verdadero negativo (TN):** el dato en la muestra es negativo y la prueba predice un negativo.
- **Falso negativo (FN):** el dato en la muestra es positivo, pero la prueba predice un negativo. También se denomina error de tipo II en estadística.
- **Falso positivo (FP):** el dato en la muestra es negativo, pero la prueba predice un positivo. También se denomina error de tipo I en estadística.

Existen algunas métricas para evaluar los distintos modelos.

La proporción de verdaderos positivos (TPR, también llamada **sensibilidad**) es la probabilidad de que un resultado positivo real dé positivo. Se calcula como $\frac{TP}{TP+FN}$.

La proporción de falsos negativos es la probabilidad de que se produzca una falsa alarma, es decir, que se dé un resultado positivo cuando el valor verdadero sea negativo. Se define como $\frac{FP}{TN+FP}$.

La proporción de verdaderos negativos (también llamada **especificidad**), es la probabilidad de que un resultado negativo real dé un resultado negativo. Se calcula como $\frac{TN}{TN+FP}$.

La precisión es la precisión general del sistema. Se define como $\frac{TP+TN}{TP+TN+FP+FN}$.

Tomado de [6].

Etapa de predicción

En esta etapa se toman los nuevos datos que se quieren clasificar y se los envía al sistema para que los clasifique. De esta manera, se logra clasificar una entrada nueva a partir de datos que sirvieron para entrenar un modelo y parametrizarlo lo mejor posible.

1.1.4 ¿Qué es la Seguridad Informática?

Entiéndase a la Seguridad Informática, en adelante SI, como la disciplina cuya tarea es la preservación de la integridad y la privacidad de la información almacenada en un sistema informático. La SI no solo hace referencia a la investigación sino también a la ejecución de políticas de protección de datos por parte de un equipo profesional. Dichas prácticas son diversas y a menudo consisten en la restricción del acceso al sistema o a partes del sistema. Estas iniciativas buscan preservar la integridad de la información, la confidencialidad y la disponibilidad. [7]

- Confidencialidad (C): la información de los sistemas solo es accedida a los usuarios autorizados.
- Integridad (I): la información de los sistemas solo puede ser creada y modificada por los usuarios autorizados.
- Disponibilidad (D): los usuarios deben tener disponible toda la información del sistema cuando así lo deseen.

Cabe mencionar que no existe ninguna técnica que permita asegurar la inviolabilidad de un sistema.

1.1.5 Enfoques basados en Machine Learning

Los esfuerzos combinados entre los algoritmos de ML sumados a la experiencia de los analistas sin duda en un futuro contribuirán a facilitar y a optimizar detección de anomalías. Hoy en día es una batalla constante entre los distintos ataques facilitados en parte por la cantidad de información disponible en internet que a diario se publica. Las técnicas de detección y prevención de ataques suelen estar por detrás de las nuevas técnicas disponibles día tras día.

Las técnicas de ML evolucionaron notablemente sumado a la gran potencia de cálculo y la incontable información disponible. Como la cantidad de información que los analistas deben analizar de distintas fuentes y formatos supera ampliamente toda capacidad humana, hacer uso de dichas técnicas potencia enormemente la clasificación y correlación de distintas fuentes. Si tuviesen que ser realizadas manualmente sería imposible. Para la detección de anomalías en la ciberseguridad son utilizados tanto los algoritmos supervisados como no supervisados. En general, es más visible la utilización de algoritmos supervisados, pero los no supervisados se usan también. [5]

Algunos ejemplos donde se utilizan estos algoritmos son por ejemplo en la detección de malware. Es bien sabido que la detección basada en firmas tiene grandes inconvenientes. Estos fallan por ejemplo al detectar *zero days*, ya que no se cuenta con firmas que permitan su detección. Otros casos donde fallan son por ejemplo con la inserción de *no-ops* y ofuscación de código, ya que ambas modifican las secuencias de bytes que son utilizadas para las firmas.

Otro ejemplo es la detección de intrusos utilizando inteligencia artificial. La intrusión en una red es un acto deliberado que se realiza para utilizar los recursos informáticos de la red y/o para amenazar la red o la información.

En el presente trabajo se analizarán ambos ejemplos como casos de estudio. Para la realización de las pruebas se utilizará un computador con procesador Intel Core i7, 32 GB de memoria RAM y disco de estado sólido. Se utilizarán *datasets* bajados de Internet con ejemplos de ambas técnicas. Se descartan los dos primeros pasos descriptos en el apartado anterior.

1.2 Definiciones de algoritmos

En esta sección se dará una introducción para los distintos algoritmos presentados durante el trabajo.

1.2.1 Support Vector Machine (SVM)

De [8] se toma como fuente para esta sección.

Para un conjunto de puntos separable por dos clases, como el mostrado en la imagen a continuación, existen varias formas de separaciones lineales.

Intuitivamente una **frontera de decisión** que se dibuja en el espacio entre ambas clases es mejor que una frontera que se aproxime demasiado a una u otra clase.

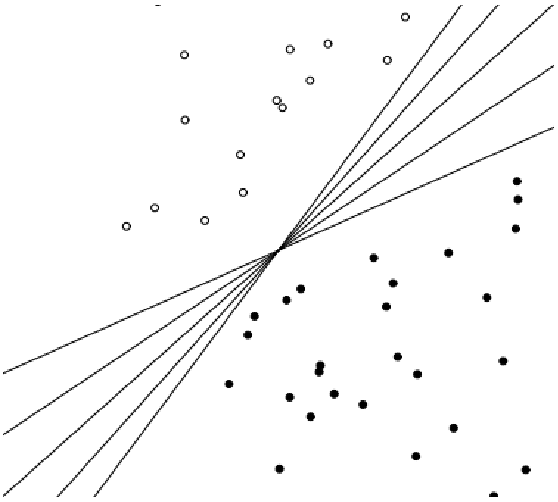

Ilustración 1 *Hay infinitos hiperplanos que separan a dos clases linealmente separables.*

SVM define en particular el criterio para encontrar aquel hiperplano que esté lo más lejos posible de cualquier punto. Esta distancia desde el hiperplano hasta el punto determina el **margen** del clasificador. Este método de construcción significa necesariamente que la función de decisión para un SVM está completamente especificada por un subconjunto de los puntos que define la posición de la frontera. Estos puntos son llamados *support vectors*.

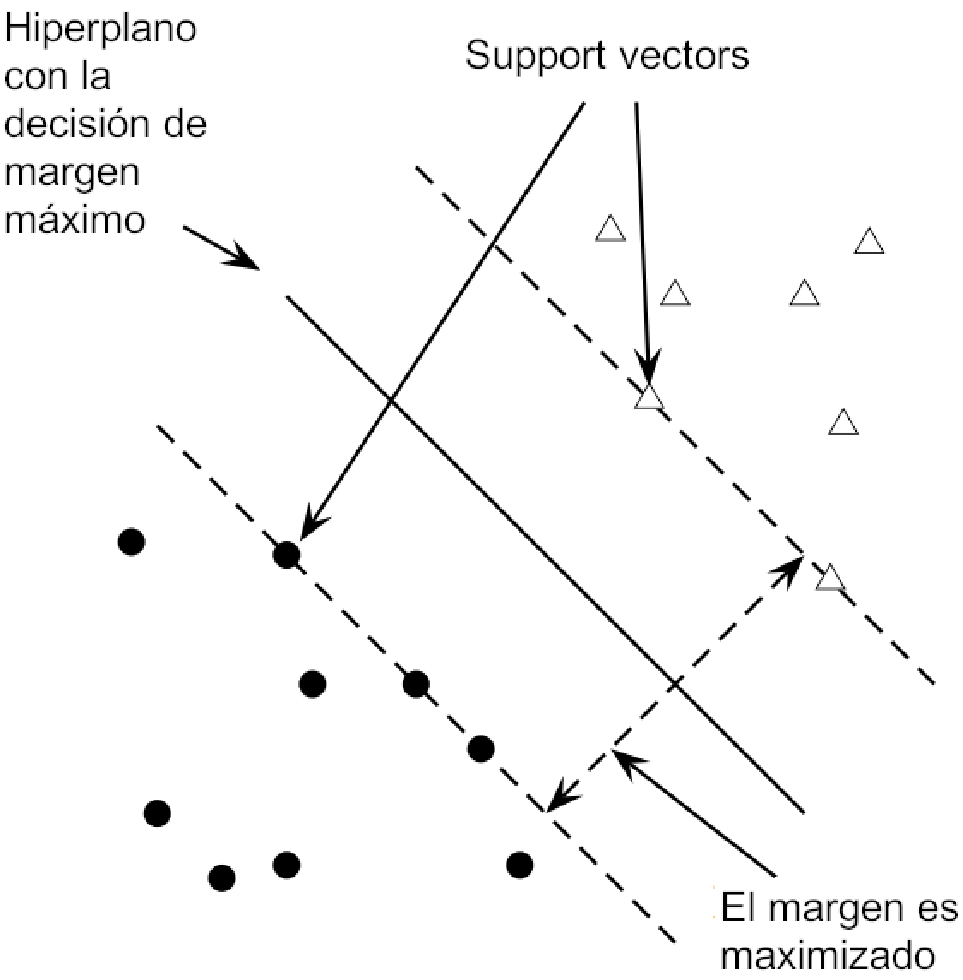

Ilustración 2 *Support vectors y el margen del clasificador*

La maximización del margen parece buena ya que aquellos puntos próximos a la frontera de decisión tendrán aproximadamente un 50% de probabilidades de pertenecer a una u otra clase. Un clasificador con un gran margen no clasifica con incertidumbre. Esto brinda un margen de seguridad en la clasificación.

Un hiperplano de decisión puede ser definido por un término de intercepción b y un vector normal al hiperplano, $\vec{w}$, definido como *vector de peso, weight vector*, en inglés. b sirve para elegir entre todos los hiperplanos que son perpendiculares al vector de peso. Todos los puntos en el hiperplano satisfacen la ecuación

$$\vec{w}^T \vec{x} = -b$$

.

Supóngase que se tienen los puntos y clases, $D = \{(\vec{x_i}, y_i)\}$ donde cada $\vec{x_i}$ es un punto y y_i la clase a la cual pertenece. Para aquellos SVM con solamente dos clases, y_i solo vale +1 o -1. El clasificador entonces es

$$f(\vec{x}) = signo(\vec{w}^T \vec{x} + b)$$

Para un dado conjunto de puntos y un hiperplano de decisión, se define el **margen funcional** del vector i-ésimo $\vec{x}_i$ con respecto al hiperplano $< \vec{w}, b >$ como

$$y_i(\vec{w}^T \vec{x_i} + b)$$

. El margen funcional de un conjunto de puntos con respecto a la superficie de decisión es dos veces el margen funcional de cualquier punto en el conjunto con margen funcional mínimo.

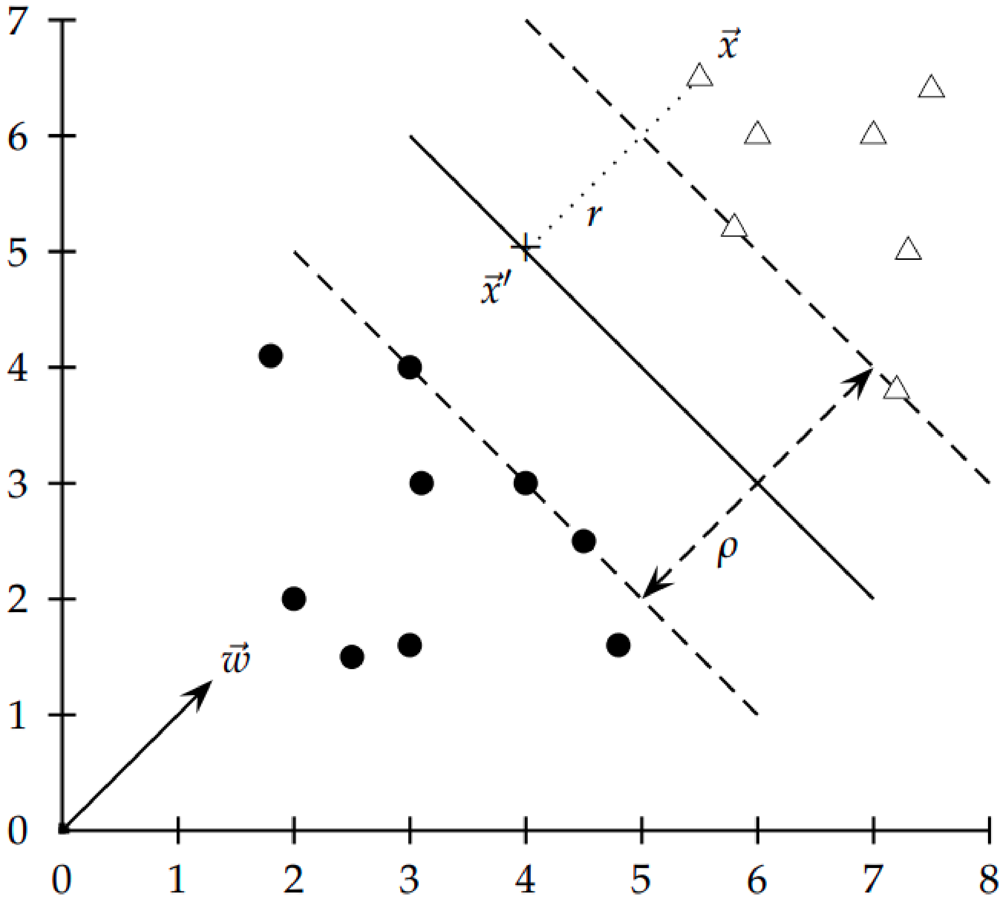

***Ilustración 3** El margen geométrico de un punto (r) y la superficie de decisión (ρ).*

En la ilustración 3, r representa la distancia euclídea del punto $\vec{x}$ a la frontera de decisión. Es sabido que la distancia más corta entre un punto y un hiperplano es perpendicular al hiperplano, y por lo tanto paralelo a $\vec{w}$. Denótese el punto más próximo a $\vec{x}$ como $\overrightarrow{x'}$. Multiplicar por y cambia el signo para los casos posibles de $\vec{x}$ estando de un lado o del otro en la superficie de decisión.

$$\vec{x}' = \vec{x} - yr\frac{\vec{w}}{|\vec{w}|}$$

Por lo tanto, se satisface la siguiente ecuación

$$w^T\vec{x}' + b = 0$$

$$\vec{w}^T\left(\vec{x} - yr\frac{\vec{w}}{|\vec{w}|}\right) + b = 0$$

Despejando r

$$r = y\frac{\vec{w}^T\vec{x} + b}{|\vec{w}|}$$

El margen geométrico del clasificador es el máximo ancho de la banda que puede ser dibujada para separar los *support vectors* de las dos clases. Esto es, dos veces el mínimo valor de los puntos para el *r* dado por la ecuación.

Por conveniencia para resolver grandes SVMs, se requiere que el margen funcional de cada punto sea al menos 1 y que sea igual a 1 para al menos un punto.

$$y_i(\vec{w}^T\,\overrightarrow{x_i} + b) \geq 1$$

El margen geométrico es $\rho = \frac{2}{\overrightarrow{|w|}}$. El objetivo es maximizar el margen geométrico sujeto a restricciones lineales. Vale mencionar que maximizar $\frac{2}{\overrightarrow{|w|}}$ es equivalente a minimizar $\frac{\overrightarrow{|w|}}{2}$.

Por lo tanto, el problema de SVM puede formularse como

Encontrar $\vec{w}$ y b tales que

$$min\ \frac{1}{2}\vec{w}^T\vec{w}$$
$$s.t.\ \forall\{(\overrightarrow{x_i}, y_i)\}, y_i(\vec{w}^T\,\overrightarrow{x_i} + b) \geq 1$$

La solución de un SVM consiste en la optimización de una función cuadrática sujeto a restricciones lineales. Se denominan algoritmos de optimización cuadrática (QP). Existen algoritmos bien conocidos y detallados que los resuelven. El tiempo de entrenamiento y testeo de SVM dependen del algoritmo utilizado para la resolución de la QP. Empíricamente, se obtiene una complejidad temporal de $O(n^3)$, donde *n* es el tamaño de conjunto de puntos.

Hasta ahora se habló únicamente de un caso linealmente separable y con dos clases. Pero eso se puede generalizar.

Como caso general los SVM no son linealmente separables. El enfoque que se utiliza para esos casos es el de permitir cometer errores, es decir, puntos que están en el lado incorrecto del margen. Entonces, se paga un costo por cada ejemplo clasificado erróneamente. Para implementar esto se hace uso de las variables *slack* ξ_i. Un valor no negativo de la variable *slack* habilita a $\overrightarrow{x_\iota}$ a no tener que cumplir con los requisitos del margen a un costo proporcional a ξ_i. La nueva ecuación queda definida por

Encontrar $\vec{w}$ y b tales que

$$min\ \frac{1}{2}\vec{w}^T\vec{w} + C\sum\nolimits_i \xi_i$$
$$s.t.\ \forall\{(\overrightarrow{x_\iota}, y_i)\}, y_i(\vec{w}^T\,\overrightarrow{x_\iota} + b) \geq 1 - \xi_i$$

El parámetro *C* es un término de regularización, el cual provee una fórmula para lidiar con el *overfitting.*

Si se desea trabajar con más de 2 clases, se procede con la construcción de SVM multiclase, donde un clasificador de dos clases es construido sobre un vector de *features* $\Phi(\vec{x}, y_i)$ derivado del par que consta de las características de entrada y la clase del dato. Durante la fase de entrenamiento, el clasificador selecciona la clase $y = arg\ max_{y'}\Phi(\vec{x}, y')$. El margen durante el entrenamiento es la diferencia entre este valor para la clase correcta y el más cercano de otra clase. Por lo tanto, la formulación cuadrática del problema va a requerir que

$$\forall i\ \forall y\ \neq y_i\ \vec{w}^T\,\Phi(\vec{x}, y_i) - \vec{w}^T\,\Phi(\vec{x}, y) \geq 1 - \xi_i$$

El caso lineal se basa en el producto interno canónico. Se define como:

$$K\left(\overrightarrow{x_\iota}, \overrightarrow{x_J}\right) = \overrightarrow{x_\iota}^T\overrightarrow{x_J}$$

Ahora supóngase que se desea mapear cada punto a un espacio de más dimensiones por medio de alguna transformación $\Phi\colon \vec{x} \mapsto \phi(\vec{x})$. Por lo tanto el producto interno se convierte en $\phi(\overrightarrow{x_\iota})^T\phi(\overrightarrow{x_J})$. Esto se denomina como *kernel trick*. No es necesario mapear $\Phi\colon \vec{x} \mapsto \phi(\vec{x})$, solo basta con calcular

$$K\left(\overrightarrow{x_\iota}, \overrightarrow{x_J}\right) = \phi(\overrightarrow{x_\iota})^T\phi(\overrightarrow{x_J})$$

Existen distintos tipos de kernels, se menciona el caso especial del kernel gaussiano

$$K(\vec{x}, z) = e^{-(\vec{x}-z)^2/2\sigma^2}$$

1.2.2 Naive Bayes

De [9] se toma la siguiente sección.

Es un modelo probabilístico que se utiliza para tareas de clasificación. Está basado en el teorema de Bayes.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Este teorema sirve para encontrar la probabilidad de un evento A sabiendo que el evento B ocurrió. B es la evidencia y A la hipótesis. Naive, ingenuo en francés e inglés, se lo llama porque se consideran a los predictores/*features* independientes.

La variable numérica y representa a la clase que se desea clasificar, mientras que la variable X representa a un vector de *n features* $X = (x_1, x_2, x_3, \dots, x_n)$.

Para este caso, el teorema de Bayes puede ser reescrito como

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

Haciendo uso de la regla de la cadena, la expresión anterior puede ser reescrita como

$$P(y|x_1, x_2, \dots, x_n) = \frac{P(x_1|y)P(x_2|y) \dots P(x_n|y)P(y)}{P(x_1)P(x_2) \dots P(x_n)}$$

$$P(y|x_1, x_2, \dots, x_n) \propto P(y)\prod_i^n P(x_i|y)$$

Por lo tanto, es necesario encontrar la clase y cuya probabilidad sea máxima. Esto puede ser aplicado a casos donde haya más de 2 clases.

$$y = argmax_y P(y)\prod_i^n P(x_i|y)$$

Con la expresión anterior, es posible obtener la clase con los predictores dados.

Existen varios tipos de clasificadores:

- Naïve Bayes Multinomial: considera un vector donde un determinado término representa la cantidad de veces que aparece.
- Bernoulli Naïve Bayes: se aplica cuando se desea determinar si una feature existe o no.
- Naïve Bayes Gaussiano: Cuando se trabaja con datos continuos, se suele asumir que los valores continuos asociados con cada clase se distribuyen de acuerdo con una distribución normal (o gaussiana). Se supone que la probabilidad de las *features* es $P(x_i|y) = \frac{1}{(2\pi\sigma_y^2)^{\frac{1}{2}}} exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$

1.2.3 *K* nearest neighbor (KNN)

Es un algoritmo supervisado que puede ser usado tanto para clasificación como para regresión. Hace uso de la *feature similarity* para predecir a qué clúster pertenecerá el nuevo punto. Es un algoritmo *lazy* ya que no aprende una función discriminativa del conjunto de entrenamiento, sino que memoriza el conjunto en su lugar. Se considera un algoritmo no paramétrico ya que no hace una suposición sobre el patrón de distribución de los datos. [10]

El funcionamiento de este algoritmo es muy sencillo: [11]

1. Elegir un parámetro k impar que indica la cantidad de clústeres que necesita encontrar. La elección de este hiper-parámetro impacta significativamente en los resultados obtenidos. También es necesario elegir una **métrica**.
2. Encontrar los k-vecinos más próximos de la muestra que se desea clasificar.
3. Asignar una clase por voto mayoritario.

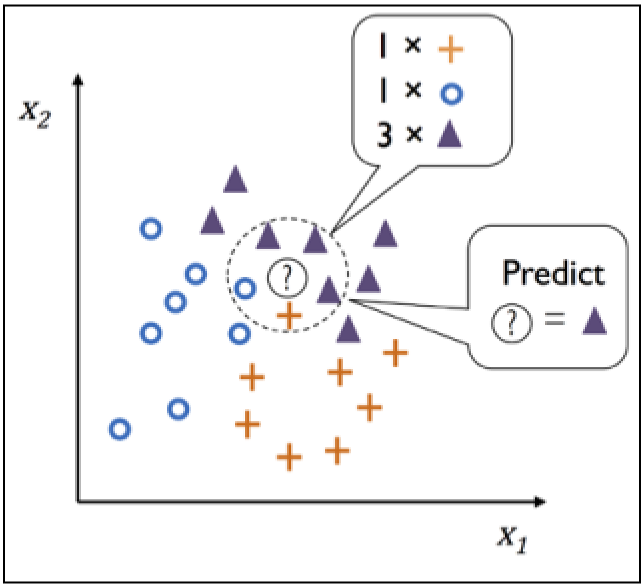

Ilustración 4 *Clasificación de un punto ? usando KNN.*

Existen distintos ejemplos de métricas para medir las distancias:

1. Distancia euclídea: $d(X,Y) = [(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2]^{1/2}$
2. Distancia Manhattan: $d(X,Y) = |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n|$
3. Distancia Chebyshev: $d(X,Y) = max(|x_1 - y_1|, |x_2 - y_2|, \dots, |x_n - y_n|)$
4. Distancia Minkowski: $d(X,Y) = [(x_1 - y_1)^p + (x_2 - y_2)^p + \cdots + (x_n - y_n)^p]^{1/p}$

1.2.4 *Decision Tree (DT)*

De [11] se toma la siguiente sección.

Es un algoritmo que desglosa los datos al tomar una decisión basada en hacer una serie de preguntas. Basado en las *features* del conjunto de entrenamiento, el algoritmo aprende una serie de preguntas para inferir las clases de las muestras.

Usando el algoritmo de decisión, comienza desde la raíz del árbol y partiendo (*splitting*) los datos por la *feature* cuya **ganancia de información** (IG) sea mayor. Este proceso iterativo se repite hasta que el procedimiento de *splitting* en cada nodo hijo dé como resultado una hoja pura. Esto significa que las muestras en cada nodo pertenecen a la misma clase.

Para poder partir los nodos en las *features* más significativas, es necesario definir la función objetivo que debe ser optimizada con el *tree learning algorithm*.

La definición de la función es la siguiente

$$IG\left(D_p, f\right) = I\left(D_p\right) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

f es la *feature* en donde partir, D_p y D_j son el conjunto de datos del nodo padre y del nodo j-ésimo, I es la medida de impureza, N_p es el número total de muestras que pertenecen al nodo padre y N_j es el número de muestras en el nodo j-ésimo. Como se puede observar en la ecuación, la IG es simplemente la diferencia entre la impureza del nodo padre y la sumatoria de impurezas de los nodos hijos. Sin embargo, por simplicidad y para reducir el espacio de búsqueda, se suelen implementar árboles de decisión binarios, donde cada nodo tiene dos nodos hijos, D_{left} y D_{right}.

Las medidas de impureza más utilizadas son la impureza de Gini (I_G), entropía (I_H) y el error de clasificación (I_E).

Se define como $p(i|t)$ a la probabilidad de que las muestras pertenezcan a la clase c para un nodo t en particular.

La entropía se define como:

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

Si la entropía da 0 entonces todas las muestras pertenecen a la misma clase y, por otro lado, la entropía es máxima si las clases siguen una distribución uniforme.

La impureza de Gini puede ser interpretada como el criterio que minimice la probabilidad de clasificar erróneamente.

$$I_G(t) = -\sum_{i=1}^{c} p(i|t)\left(1 - p(i|t)\right) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Como sucede con la entropía, la impureza de Gini es máxima si las clases están perfectamente mezcladas. En la práctica, ambas medidas dan resultados muy similares. Por último, queda el error de clasificación:

$$I_E = 1 - max\{p(i|t)\}$$

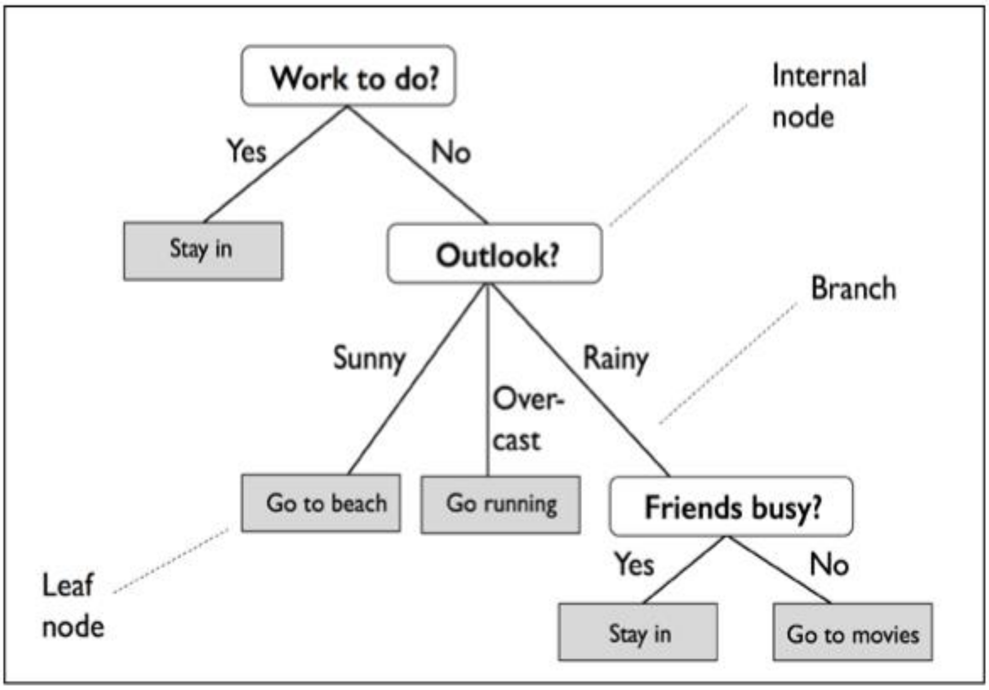

Ilustración 5 *Ejemplo de Decision Tree.*

1.2.5 Random Forest

De [11] se toma la siguiente sección.

Intuitivamente, Random Forest puede ser considerado como un conjunto de árboles de decisión. La idea que está detrás de Random Forest es la de promediar múltiples árboles de decisión que individualmente tienen mucha varianza. El objetivo es la construcción de un modelo robusto que tenga mejor performance de generalización y que sea menos susceptible al *overfitting*.

No ofrecen el mismo nivel de facilidad de interpretación que los DT, pero tienen la gran ventaja de que no hay que preocuparse en seleccionar un buen valor para el hiper-parámetro. El único hiper-parámetro que hay que seleccionar es el número de árboles k. Por lo general, a mayor k mayor calidad en la clasificación a mayor costo computacional.

El algoritmo puede resumirse en cuatro simples pasos:

1. Dibujar una muestra de arranque aleatoria de tamaño n.
2. Hacer crecer un árbol de decisiones a partir de la muestra de arranque. En cada nodo:

    a. Seleccionar aleatoriamente d funciones sin reemplazo.
    b. Dividir el nodo utilizando la función que proporcione la mejor división de acuerdo con la función objetivo, por ejemplo, maximizando la ganancia de información.
3. Repetir los pasos 1-2 k veces.
4. Agregar la predicción por cada árbol para asignar la etiqueta de clase por mayoría de votos.

1.2.6 *Stochastic Gradient Descent (SGD)*

De [11] se toma la siguiente sección.

Uno de los elementos clave de los algoritmos supervisados de *machine learning* es la optimización de una función objetivo durante el proceso de aprendizaje. Por lo general, se busca minimizar esta función objetivo.

$$J(w) = \frac{1}{2}\sum_{i}\left(y^{(i)} - \phi\left(z^{(i)}\right)\right)^{2}$$

Se puede definir una función J para aprender los pesos como la sumatoria de los errores cuadráticos entre la clase calculada y la real. La función ϕ se denomina función de activación lineal.

El $^{1}/_{2}$ se agrega para simplificar la posterior derivación. La ventaja principal de esta función es que es diferenciable. A su vez, es convexa, por lo tanto, puede ser optimizada con el algoritmo **gradient descent** para encontrar aquellos pesos que minimicen la función de costo.

Definición 1: *(función convexa): Sea* I *un intervalo no vacío de* $\mathbb{R}$*. Una función* $f: I \to \mathbb{R}$ *se dice que es convexa sobre* I *si cumple* $f(\alpha x + (1-\alpha)x') \leq \alpha f(x) + (1-\alpha)f(x')$

El funcionamiento de este algoritmo consiste en que, en cada iteración, se toma un paso en dirección contraria al gradiente, donde dicho paso está determinado por el valor de la tasa de aprendizaje (l*earning rate*). Esto se repite hasta que se alcanza un mínimo local o global.

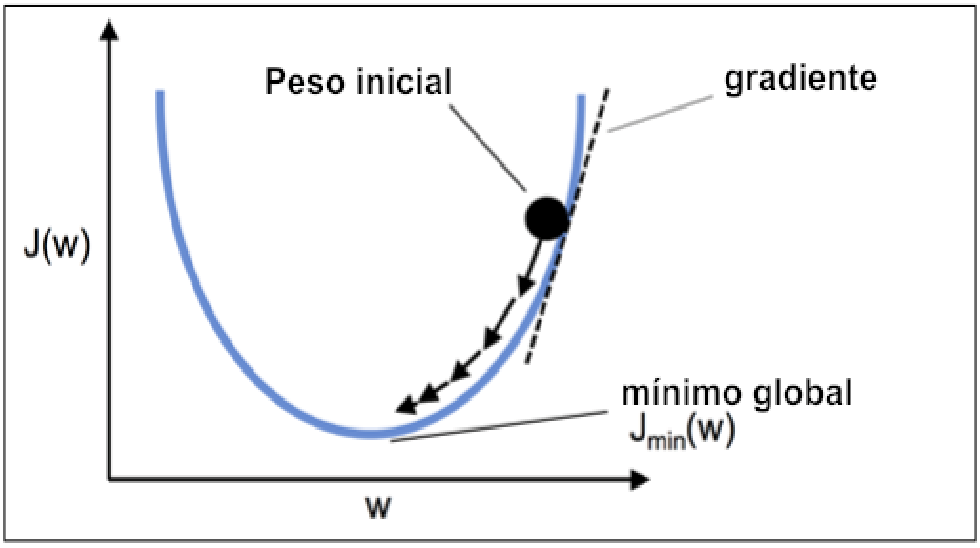

Ilustración 6 *Gradient Descent.*

$$w := w + \Delta w$$

Donde el cambio del peso Δw está definido como el gradiente negativo multiplicado por la tasa de aprendizaje (η).

$$\Delta w = -\eta \nabla J(w)$$

Para el cálculo del gradiente, hay que calcular las derivadas parciales de la función costo respecto a cada peso w_j:

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

La actualización del peso w_j puede ser escrita como:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = -\eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Los algoritmos mencionados anteriormente no escalan cuando se trata de muchos datos, por lo tanto, necesita algunas modificaciones. Algunos autores, llaman al método anterior como **batch gradient descent (BGD)**.

Una alternativa es usar **stochastic gradient descent**, que en lugar de actualizar los pesos con la suma de los errores cuadráticos, los pesos se actualizan incrementalmente para cada muestra:

$$\Delta w = \eta \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x^{(i)}$$

SGD suele converger mucho más rápido que con BGD debido a las actualizaciones de peso más frecuentes. También puede tener la ventaja de que el SGD puede escapar de los mínimos locales poco profundos más fácilmente si se trabaja con funciones de costo no lineales.

1.2.7 *KMeans*

De [11] se toma la siguiente sección.

Es un algoritmo sencillo y eficiente a la vez en comparación con otros algoritmos para *clustering*. Pertenece a la clase de algoritmos no supervisados. Se define centroide como el punto representativo del clúster.

Una desventaja de este algoritmo es que es sensible a la elección del hiper-parámetro k. Una elección incorrecta de dicho valor puede ocasionar una pobre performance.

Puede ser resumido en los siguientes pasos:

1. Elegir aleatoriamente k centroides de los puntos del conjunto como clústeres iniciales.
2. Asignarle a cada muestra el centroide más próximo $\mu^{(j)}, j \in \{1, \dots, k\}$
3. Mover los centroides al centro de las muestras que le fueron asignados.
4. Repetir 2 y 3 hasta que los clústeres no tengan modificaciones o se alcance un número máximo de iteraciones.

La cuestión clave es cómo medir la similitud entre objetos. Basado en la distancia euclídea como métrica, es posible definir a k-means como un problema simple de optimización. En cada iteración se busca la minimización de la suma de errores cuadráticos (SSE).

$$SSE = \sum_{i=i}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| x^{(i)} - \mu^{(j)} \right\|^2$$

Donde $\mu^{(j)}$ es el centroide para el cluster j, y $w^{(i,j)} = 1$ si la muestra $x^{(i)}$ está en el clúster j; $w^{(i,j)} = 0$ en caso contrario. Es importante recalcar que los datos tienen que estar normalizados, ya que este algoritmo es sensible a los valores que no están en una misma escala.

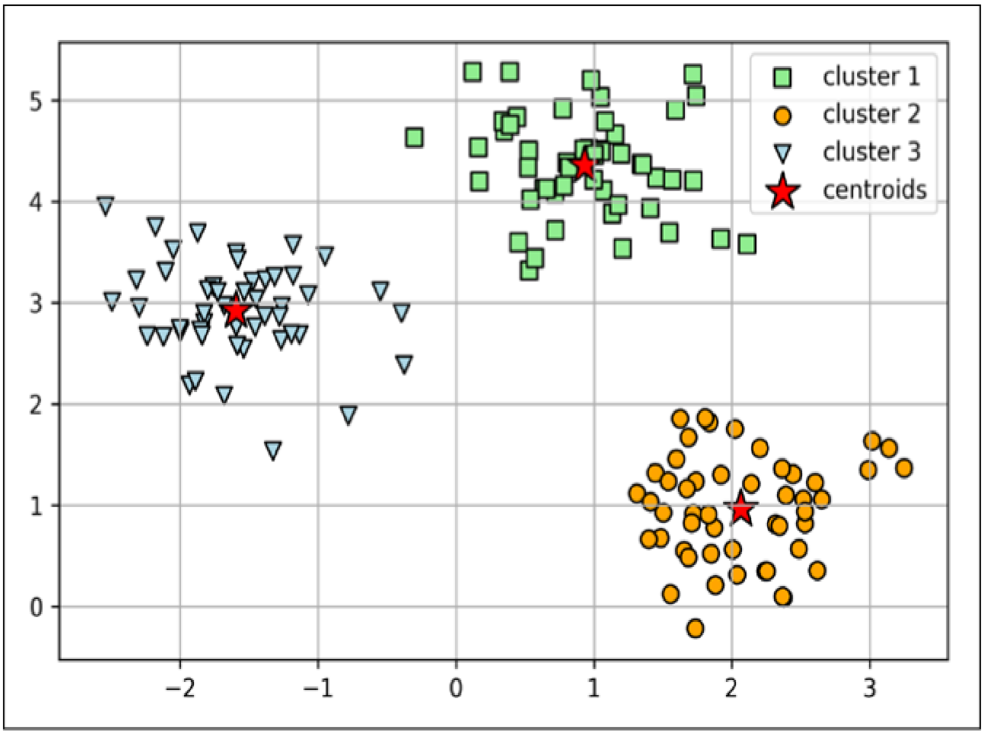

Ilustración 7 *k-means.*

El algoritmo clásico ubica los centroides de manera aleatoria. Esto puede resultar en clústeres que no son buenos o que converja lento. Para ello, hay otro algoritmo basado en k-means llamado k-means++ que ofrece mejores resultados que el k-means original. La diferencia radica en cómo ubica los centroides iniciales.

1.2.8 DBSCAN

De [11] se toma la siguiente sección.

En DBSCAN la noción de densidad está definida como el número de puntos dentro de un radio ε. Según el algoritmo, una etiqueta especial es asignada a cada muestra (punto) utilizando los siguientes criterios:

- Un punto es considerado un punto principal (*core point*) si al menos un número especificado de puntos vecinos (MinPts) caen dentro del mismo radio ε.

- Un punto de frontera (*border point*) es uno que tiene menos vecinos que MinPts dentro de ε, pero se encuentra dentro del radio ε de un punto central.
- Todos aquellos puntos que no son ni principales ni frontera son considerados ruido (*noise points*).

Luego de haber etiquetado los puntos como *core*, *border* o *noise*, el algoritmo se puede resumir en dos simples pasos:

1. Formar un grupo separado para cada punto central o grupo conectado de puntos principales
2. Asignar cada punto de borde al clúster de su punto principal correspondiente.

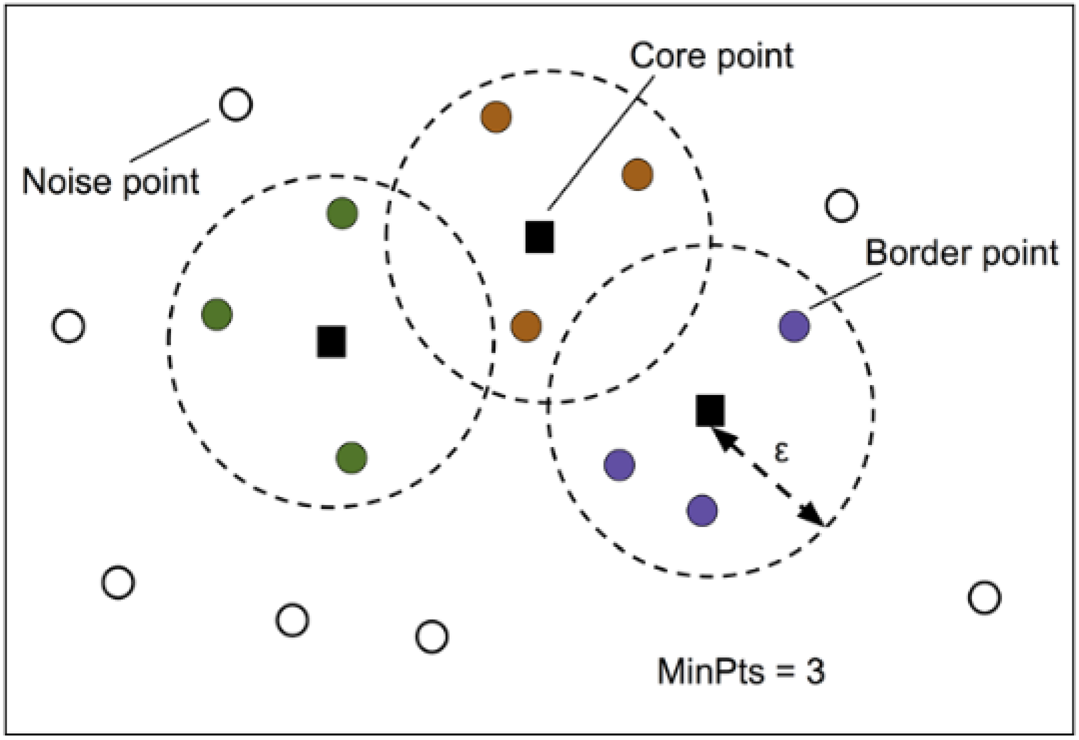

Ilustración 8 *DBSCAN*

Una de las ventajas principales que ofrece en comparación con k-means es que no asume que los clústeres son esféricos. Además, no necesariamente asigna cada punto a un clúster y es capaz de remover aquellos puntos ruidosos.

1.2.9 PCA

De [11] se toma la siguiente sección.

Es una transformación lineal que pertenece a los algoritmos no supervisados. Mayoritariamente se utiliza para extracción de *features* y para reducción de la dimensionalidad.

Ayuda a identificar patrones en los datos basado en la correlación entre *features*. En pocas palabras, PCA tiene como objetivo encontrar las direcciones de varianza máxima en datos de alta dimensión y lo proyecta en un nuevo subespacio con dimensiones iguales o menores que el original. Es importante mencionar que PCA es sensible a la escala de los datos, por lo tanto, es necesario normalizarlos antes de ejecutar PCA.

En resumen, el algoritmo PCA puede ser descripto como:

1. Estandarizar el *dataset* de d dimensiones.
2. Construir la matriz de covarianza.
3. Calcular los autovalores y autovectores de la matriz de covarianza.
4. Ordenar los autovalores en orden decreciente.
5. Seleccionar los k autovectores que corresponden con los k autovalores mayores. k es la dimensión del nuevo subespacio de *features (*$k \leq d$*).*
6. Construir la matriz de proyección W con los k autovectores "mayores".
7. Transformar el conjunto de datos de entrada de dimensión d utilizando la matriz de proyección W para obtener el nuevo subespacio de *features* de dimensión k.

1.2.10 LDA

De [11] se toma la siguiente sección.

El concepto de LDA está muy relacionado con PCA.

PCA intenta encontrar los componentes de máxima varianza, mientras que el objetivo de LDA es encontrar el subespacio de *features* que optimicen la separación de clases. LDA es un algoritmo que pertenece a la clase de supervisados y puede ser utilizado para reducir dimensiones.

En resumen, el algoritmo PCA puede ser descripto como:

1. Estandarizar el *dataset* de d-dimensiones, donde d es el número de *features*.
2. Para cada clase, calcular el vector de medias de d-dimensiones.

3. Construir la matriz de dispersión entre clases S_B y la matriz de dispersión dentro de la clase S_W.
4. Calcular los autovalores y autovectores de la matriz $S_W^{-1}S_B$
5. Ordenar los autovalores en orden decreciente
6. Seleccionar los k autovectores que corresponden con los k autovalores mayores para construir la matriz de transformación W. Los autovectores son las columnas de la matriz.
7. Proyectar las muestras hacia el nuevo subespacio de *features* usando la matriz de transformación W.

Capítulo 2: Machine Learning aplicado a Sistemas de Detección de Intrusos

2.1 Introducción

2.1.1 Tráfico de red

El tráfico de red son aquellos paquetes que son enviados de un origen a un destino dentro de una red. La arquitectura de red está separada en capas donde en cada una operan distintos protocolos para realizar tareas específicas. El modelo OSI describe siete capas que van desde la física, la primera capa, hasta la capa de aplicación que es aquella con la que interactúa el usuario. Los protocolos más usados en la capa cuatro son UDP y TCP. La diferencia entre ellos radica en que TCP está orientado a la conexión, esto significa que se garantiza la recepción de paquetes de ambas partes. Por otro lado, UDP simplemente envía paquetes sin tener en consideración si la contraparte recibió el paquete, es decir, no toma en cuenta la pérdida de los mismos. Este tipo de paquetes son útiles por ejemplo para el video y audio sobre Internet, así también para el *streaming.*

2.1.2 Tráfico de red malicioso

El tráfico de red malicioso es aquel cuyo objetivo es generar daño o una intrusión en alguna computadora, sistema o red. [12]

2.1.3 Caracterización del tráfico basado en flujos

Un flujo se define como un conjunto de paquetes que tiene un ID conformado por una tupla formada por IP origen, IP destino, puerto origen, puerto destino y protocolo. Los flujos son bidireccionales, esto significa que el primer paquete del flujo determina el origen. [12]

Está probado que la caracterización basada en flujos es mucho más eficiente y ofrece una buena performance de detección en comparación con la inspección de paquetes. Este último es un proceso que consume demasiados recursos y está limitado a aquellos paquetes que no viajen encriptados. Como contrapartida, la clasificación de paquetes basada en inspección ofrece mayor precisión que la caracterización basada en flujos.

2.1.4 Sistemas de Detección de Intrusos

Un sistema de detección de intrusos (IDS por sus siglas en inglés) no solamente detecta intrusiones, sino que también es capaz de detectar otros tipos de tráfico malicioso, ya sea ataques de denegación o escaneos de puertos. Básicamente los IDS consisten en tres componentes lógicos: [12]

- Sensores: recolecta y decodifica los paquetes para luego enviarlos al analizador.
- Analizadores: son los encargados de detectar tráfico sospechoso y realizar una acción.
- Interfaz de usuario: muestra la salida del IDS y se puede utilizar para configurar el IDS.

En el mercado hay varios de estos sistemas, tanto comerciales como *open-source*. Por mencionar algunos comerciales están *Juniper*, *Cisco*, *Symantec*, *McAfee*, etc. Por otro lado, están los *open-source*, entre ellos, *Suricata*, *Snort* y *Zeek*. El más antiguo es *Snort*, en activo desarrollo desde 1998. En comparación con *Snort*, *Suricata* es más rápido y tiene mejor desempeño. Esto se debe a que *Suricata* está hecho con técnicas de programación multihilo, mientras que *Snort* corre sobre un único hilo de ejecución. Ambos trabajan utilizando un conjunto de reglas que el usuario puede adaptar a sus necesidades. El problema común que tienen estos IDS es que disparan falsas alarmas. Esto constituye un problema ya que pueden llegar a bloquear tráfico benigno así también como la mal utilización de recursos informáticos. [13]

2.2 Técnicas de Machine Learning aplicadas a la detección de tráfico malicioso

2.2.1 Dataset utilizado

Para realizar las siguientes pruebas se va a utilizar el *dataset* CICIDS2017[4]. Este *dataset* es provisto por la Universidad de Nuevo Brunswick en Canadá.

2.2.2 Preprocesamiento del dataset

Como se mencionó anteriormente antes de comenzar a trabajar con el *dataset* en cuestión es necesario preprocesarlo. En una primera instancia, se transformaron las distintas etiquetas que se esperaban como resultados de la clasificación. Luego, se quitaron los valores fuera de rango. Este *dataset* no contaba con valores nulos.

Se generaron dos conjuntos de datos, uno de entrenamiento y otro de prueba. El de entrenamiento fue del 70% de los datos, mientras que el 30% restante era para el testeo.

Se ejecutó el algoritmo de selección de *features* LDA para reducir la dimensionalidad de los datos.

2.2.3 Comparación de resultados obtenidos con distintas técnicas

Para la comparación de la performance de los distintos algoritmos de ML, se compararon, utilizando los mismos *sets* de entrenamiento y testeo. Se evaluó también el desempeño del algoritmo con reducción de *features* y con normalización de datos.

También, se midió el tiempo de ejecución de cada algoritmo y se calculó la matriz de confusión.

Los algoritmos que se evaluaron fueron:

- KNN con 15 clases, correspondientes a las 15 categorías que contiene el *dataset*.

---

[4] https://www.unb.ca/cic/datasets/ids-2017.html

- Naïve Bayes Gaussiano
- Stochastic Gradient Descent (SGD)
- Decision Tree

Para los siguientes casos se utilizó LDA (*Linear Discriminant Analysis*) para maximizar la separación de las clases. Se pudo verificar que LDA obtiene mejores resultados con 2 componentes.

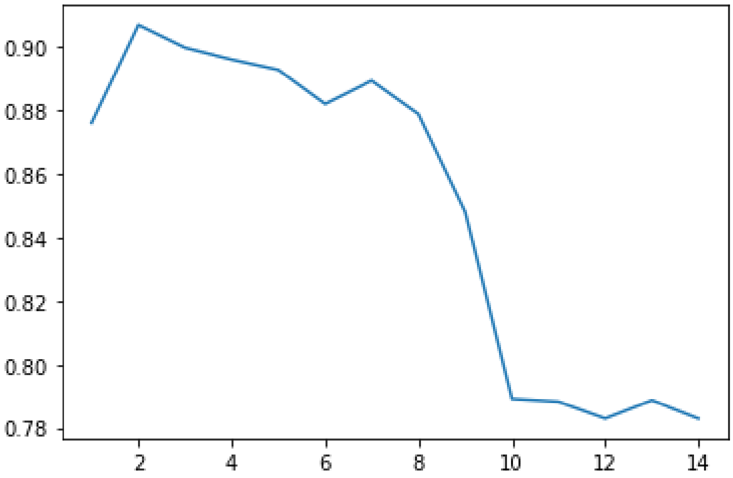

Ilustración 9 *Evaluación de dimensiones de LDA utilizando Naïve Bayes Gaussiano.*

Tabla 1 *Ejecución de KNN con 15 features*

	Con *feature reduction*	Sin *feature reduction*	Sin *feature reduction* (normalizado*)*
Performance (precisión)	0.9924943153918346	0.9936483114564726	0.9971220751204336
Tiempo de ejecución	29.573192596435547	997.5887162685394	5995.692269086838

Tabla 2 *Ejecución de Naïve Bayes Gaussiano*

	Con *feature reduction*	Sin *feature reduction*	Sin *feature reduction* (normalizado*)*
Performance (precisión)	0.9066440734648026	0.1216935952040865 5	0.2364337753452273 3
Tiempo de ejecución	0.7987778186798096	7.28095006942749	5.76328182220459

Tabla 3 *Ejecución de Decision Tree*

	Con *feature reduction*	Sin *feature reduction*	Sin *feature reduction* (normalizado*)*
Performance (precisión)	0.9924848950157968	0.9986022517053824	0.9986034292523872
Tiempo de ejecución	10.555210828781128	158.20431661605835	142.65307211875916

Tabla 4 *Ejecución de SGD con regresión logística y 50 iteraciones*

	Con *feature reduction*	Sin *feature reduction*	Sin *feature reduction* (normalizado*)*
Performance (precisión)	0.9020728359924307	0.8505893033985185 (*)	0.9367398198117574
Tiempo de ejecución	28.653868675231934	268.540034532547 (*)	102.8610577583313

(*) Se alcanzó el número máximo de iteraciones.

Para los siguientes casos se utilizó PCA (*Principal Component Analysis*) para reducir la cantidad de dimensiones. La base son las componentes principales.

Se pudo verificar que el estimador de máxima verosimilitud[5] (MLE por sus siglas en inglés) ofreció los resultados óptimos en comparación con el número de dimensiones. Se puede observar en el gráfico a continuación que con un clasificador Naïve Bayes Gaussiano y 1 dimensión se consiguió

---

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

alrededor de un 80% de precisión, mientras que con el MLE se consiguió aproximadamente un 90,6% de precisión.

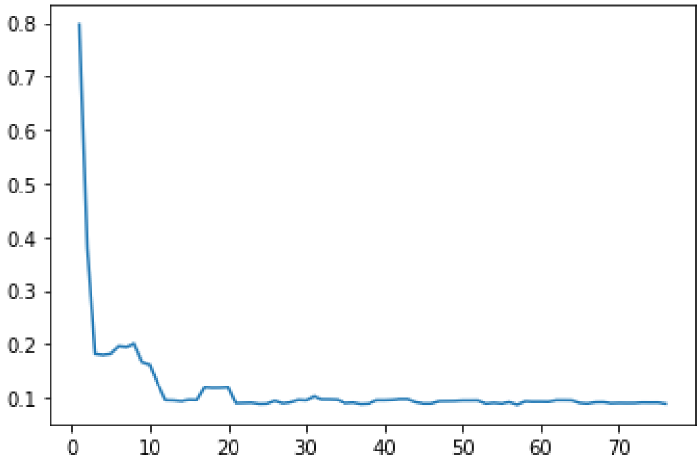

Ilustración 10 *Evaluación de dimensiones de PCA utilizando Naïve Bayes Gaussiano.*

Tabla 5 *Ejecución de KNN con 15 features*

	Con *feature reduction*
Performance (precisión)	0.9066440734648026
Tiempo de ejecución	0.6793093681335449

Tabla 6 *Ejecución de Naïve Bayes Gaussiano*

	Con feature reduction
Performance (precisión)	0.9066440734648026
Tiempo de ejecución	0.6793093681335449

Tabla 7 *Ejecución de Decision Tree*

	Con *feature reduction*
Performance (precisión)	0.9924848950157968
Tiempo de ejecución	9.034826278686523

Tabla 8 *Ejecución de SGD con regresión logística y 50 iteraciones*

	Con *feature reduction*
Performance (precisión)	0.9089108514489127
Tiempo de ejecución	25.29490852355957

2.2.4 Conclusiones del caso de estudio

Se puede decir que la relación costo-beneficio entre los distintos clasificadores el que mejor resultado ofrece es la utilización de Decision Tree (DT). Incluso no es necesario la normalización y/o reducción de dimensiones, ya que este algoritmo no es sensible a los valores fuera de rango. Por otro lado, sí se observa que la reducción de dimensiones ofrece una clasificación en menor tiempo. Pero sabiendo que aproximadamente la reducción PCA con MLE tarda unos 36.1682 segundos, por lo tanto, conviene indudablemente basados en los datos empíricos, realizar la reducción y luego clasificar con DT.

Capítulo 3: Machine Learning aplicado a Detección de malware

3.1 Introducción

En los últimos años hubo un aumento en la investigación y en el desarrollo de soluciones que utilizan *machine learning*. Esto fue posible gracias a varios factores como:

1. El incremento en la disponibilidad de muestras rotuladas de malware disponibles para la comunidad. El objetivo de dichas muestras es facilitar la disponibilidad no solo a la comunidad de seguridad sino también para la investigación.
2. Otro factor clave fue el aumento de la capacidad de procesamiento y reducción de costos en las nuevas computadoras, haciendo así posible el uso para investigadores independientes. Esto a su vez agiliza el entrenamiento y el trabajo con volúmenes de datos más grandes.
3. El campo de *machine learning* tuvo notables avances en los últimos años, desde la mejora y desarrollo de nuevos y mejores algoritmos, lográndose así mejoras en la precisión y la escalabilidad.

Para lograr el análisis de distintas muestras de malware para luego ser clasificadas con distintas técnicas es fundamental el preprocesamiento. El objetivo de esto es extraer las *features* que servirán como entrada para los distintos algoritmos. Existen dos grandes enfoques de clasificar las *features*: estático y dinámico. Cada uno con sus ventajas y desventajas. En este trabajo solo se abordará el primero.

3.2 Análisis estático

El análisis estático consiste en la extracción de *features* sin correr el ejecutable. En el caso de estudio de los programas que corren sobre Windows, se tiene el formato PE[6]. Estas técnicas se pueden extender a otros sistemas operativos y a otras arquitecturas. Para extraer el código en ensamblador, se utilizan herramientas como IDA Pro[7] (propietario) y/o radare2[8] (gratuito, *open-source*).

Dentro de este gran subconjunto se tienen distintas técnicas:

- Análisis de strings
- N-gramas de bytes
- N-gramas de opcodes
- API Function Calls
- Basados en entropía
- Basados en imágenes
- Function Call Graph (FCG)
- Control Flow Graph (CFG)
- Análisis de PE

3.2.1 Análisis de strings

De [14] se toma la siguiente sección.

Este tipo de análisis consiste en la extracción de los strings que están embebidos dentro del ejecutable. La búsqueda de *strings* es el método más simple para descubrir cuál es la funcionalidad del programa. Para extraer se puede utilizar el comando *strings.*

Como desventaja de este método es que es altamente sensible a la ofuscación y al *packing*. Técnicas muy empleadas para dificultar el análisis y/o detección del malware. Como ventajas, es el método más simple.

Se pueden extraer URLs, nombres de archivos, claves de registro, nombres de opciones de menú, entre otras. Una vez obtenidas las cadenas, se puede ejecutar un clasificador como SVM.

---

[6] https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
[7] https://www.hex-rays.com/ida-pro/
[8] https://rada.re/n/

3.2.2 N-gramas de bytes y opcodes

Un n-grama es una secuencia continua de n elementos provenientes de una secuencia de texto. Pueden ser extraídos de secuencias de bytes representando el contenido binario del malware o desde el código en ensamblador. Tratando al archivo como una secuencia de bytes, los n-gramas son extraídos observando las combinaciones únicas de cada n-ésimo byte consecutivo. [14]

Variantes desconocidas de *malware* pueden ser reconocidas efectivamente usando n-gramas extraídos del código y de fragmentos de texto. [15]

Se demostró que los mejores resultados se obtuvieron tomando n-gramas de longitud 2.

Los n-gramas no son una *feature* adecuada para detectar código polimórfico y metamórfico debido a la sensibilidad al orden. [16]

.

3.2.3 Function API Calls

De [14] se toma la siguiente sección.

Las APIs y sus llamadas son métodos para discriminar *features*. La invocación de las llamadas a la API sirve para modelar el funcionamiento del programa. Las APIs y las *system calls* (*syscalls*) están relacionadas con los servicios que ofrecen los sistemas operativos. No existen otros métodos que no sean los mencionados para acceder a los recursos del sistema.

Existen varios ejemplos para clasificación utilizando los patrones en las llamadas a las funciones. Por ejemplo, se puede analizar el formato PE y extraer las llamadas a la API de Windows. Luego, se pueden reducir el vector de features y finalmente se puede clasificarlo utilizando un Random Forest (RF).

3.2.4 Análisis basado en entropía

De [14] se toma la siguiente sección.

La entropía de una secuencia de bytes que representa la variación estadística. Una entropía de 0 representa que el mismo carácter o secuencia fue repetido sobre el segmento analizado. La entropía está dada por la siguiente fórmula:

$$H(x) = -\sum_{i=1}^{n} P(x_i)\log_2(P(x_i))$$

Como uno de los objetivos de los creadores de malware es evitar la detección, emplean técnicas como la ofuscación y el *packing*. El análisis de la entropía se utiliza para detectar aquellos segmentos de código que fueron comprimidos y/o cifrados. Ellos cuentan con una entropía mayor al código nativo.

Como resultado, el cálculo de la media de una alta entropía es indicativa de la presencia de cifrado y/o compresión. Pero, existe malware que utiliza grandes secuencias de *nop* en afán de reducir la entropía total.

3.2.5 Análisis basado en imágenes

Los archivos ejecutables pueden ser convertidos a imágenes en escala de grises. Las imágenes de varias muestras de una misma familia de malware tendrían una representación gráfica muy similar. Esto se debe en mayor medida al código reutilizado y porque los ejecutables nuevos están creados en base a códigos de versiones anteriores. Representando los ejecutables de esta manera, sería posible detectar pequeñas variaciones entre muestras que pertenecen a una misma familia. [14]

Para convertir un binario en una imagen en escala de grises, las secuencias de bytes se representan como una imagen PNG en escala de grises. Por lo general, se toma una imagen con ancho arbitrario y longitud variable en función del tamaño del ejecutable. Luego, una vez convertido, se puede utilizar clasificadores como k-NN. [17]

Como desventaja de este método radica en cómo las imágenes son generadas. Por una lado, los ejecutables no son imágenes en 2D y al trasformarlas se le introducen suposiciones innecesarias. Por otro lado, para la construcción de la imagen, se necesita especificar un hiper-parámetro *ancho* que debe ser optimizado. Además, se introducen correlaciones espaciales que no existen y que muchas veces pueden no ser ciertas.

Por último, es sensible a la ofuscación de código, en particular, al cifrado y compresión, que alteran completamente la estructura binaria y por lo tanto pueden ser clasificados incorrectamente.

3.2.6 Análisis basado en Function Control Graph (FCG)

Un FCG es un grafo dirigido cuyos vértices representan las funciones con las cuales el programa está construido y las aristas dirigidas representan a llamadas entre aquellas funcione. Un vértice puede estar representado por alguno de los siguientes tipos de funciones:

- Funciones locales implementadas por el programador que hacen tareas específicas.
- Funciones externas que son provistas por el SO y/o librerías externas.

Definición 2: *(Call Graph): Un call graph es un grafo dirigido* G *cuyo conjunto de vértices* $V = V(G)$ *representa a las funciones y el conjunto de aristas* $E = E(G)$ *donde* $E(G) \subseteq V(G) \times V(G)$ *en correspondencia con las llamadas a funciones.*

Los *Call Graphs* son generados a partir del archivo ejecutable utilizando análisis estático con herramientas de desensamblado. En primer lugar, se utilizan técnicas para desempaquetar el programa y luego si es necesario, desencriptar el ejecutable. Luego, se utiliza un desensamblador para identificar las funciones y asignarles un nombre simbólico. Es importante mencionar que las funciones locales hechas por quien escribió el programa, no van a poder ser recuperadas, ya que se pierden en el proceso de compilación. Por lo tanto, se le asignan nombres aleatorios y únicos. En el caso de las funciones externas, los nombres van a poder ser recuperados ya que están en la *Import Address Table* (IAT). Finalmente, cuando todos los vértices son identificados, se procede con el agregado de aristas. [18]

Este tipo de representación preserva la información estructural en el código malicioso en forma de funciones y las relaciones de invocador-invocado entre ellas. [19]

Una particularidad de este tipo de grafos es que solamente las funciones locales pueden invocar a las funciones externas y no a la inversa. [17]

Una posible técnica para agrupar muestras similares y así generar mejores técnicas de detección es agrupar los FGC en función de la similitud. Para comparar los grafos, se pueden tomar de a pares, calcularles el *graph edit distance* (GED, por sus siglas en inglés) y buscar aquellos pares que minimicen el GED. Luego, con la métrica definida se demostró que se pueden agrupar utilizando el algoritmo DBSCAN. [18]

La habilidad para reconocer familias comunes de malware habilitaría a los antivirus a detectar nuevas versiones del malware basado en familias existentes.

Definición 3: *(Graph matching): Dados dos grafos,* G *y* V*, de igual orden, el problema de graph matching consiste en encontrar una biyección* $\phi: V(G) \rightarrow V(H)$ *que optimice una función de costo que mide la calidad del matching.*

Definición 4: *(Graph edit distance): Es la mínima cantidad de operaciones elementales que son necesarias para transformar un grafo* G *en otro grafo* H*. El costo está definido por cada operación de edición, donde el costo total de transformar* G *en* H *iguala a la distancia de edición.*

La métrica GED depende de la elección de las operaciones de edición y el costo asociado a cada operación. Se consideran únicamente operaciones de inserción/borrado de vértices y aristas y renombrado de vértices. El problema GED se clasifica como NP-Completo. [18]

Como la solución exacta del problema GED es computacionalmente cara, está demostrado que el método más rápido y preciso es una modificación de *Simulated Annealing (SA)*, un método de busca local que busca un *graph mapping* que minimiza el GED. [20]

Otro método para la agrupación de los grafos es utilizando la técnica Locality Sensitive Hashing (LSH) para computar una GED aproximada.

Definición 5*: (Locality Sensitive Hashing): LSH hace referencia a familias de algoritmos cuyo objetivo es encontrar un hash de los puntos de manera tal que* *los puntos cercanos entre sí estén en el mismo bucket con una alta probabilidad. Mientras que aquellos que sean más lejanos estén en buckets diferentes.*

Definición 6*: (Simulated Annealing): es una técnica probabilística para aproximar el óptimo global de una función dada.*

3.2.7 Análisis basado en Control Flow Graph (CFG)

Un CFG es un grafo dirigido en donde los nodos representan bloques básicos y las aristas representan los caminos de flujos de control. Un bloque básico es una secuencia lineal de instrucciones que tienen un punto de entrada común, es decir, la primera instrucción ejecutada. Un CFG es una representación de todos los flujos posibles que pueden ser recorridos durante la ejecución del programa. [14]

Una forma de poder convertir un archivo ejecutable en un CFG es desensamblarlo, aplicarle un algoritmo de preprocesamiento que genere los CFG en un vector de *features*, y por último usar algún clasificador (SVM o Decision Tree). Durante el preprocesamiento se capturan las llamadas a la API y son mapeadas con un número único global. De esta manera se construye el grafo de llamadas a la API.

Los pasos más importantes son el de la generación de *features* y el de selección. Como el grafo de llamadas a la API es un grafo disperso, puede ser representado como una matriz dispersa que fácilmente puede ser convertida a un vector. [20]

Definición 7: *(grafo disperso): Un grafo G es un grafo con una poca cantidad de aristas, cercano al que tendría si fuera un grafo vacío.*

Definición 8: *(matriz dispersa): Es una matriz de gran tamaño en la que la mayor parte de sus elementos son cero.*

3.2.8 Análisis de PE

Uno de los mejores enfoques de este tipo de análisis es el desarrollado por SAVE (*Static Analyzer of Vicious Executables*) que se basa en la detección por firmas de *malware* polimórfico usando una medida de similitud entre los *malwares* conocidos y el código sospechoso. La idea de este enfoque es la de extraer las firmas del malware original con la hipótesis de que todas las versiones de la misma familia comparten una firma común. Este trabajo mejoró la detección de código polimórfico, pero falla al detectar *malware* desconocido. [21]

Cuando cualquier PE es cargado en memoria, uno de los trabajos del Windows *loader* es el de localizar a todas las funciones importadas y asegurar que dichas direcciones estén disponibles al archivo que está siendo cargado. Dentro del archivo PE, existe un vector donde cada elemento tiene el nombre de la DLL importada y apunta a un vector de punteros a función. Este último es conocido como IAT (*Import Address Table*) y contiene las direcciones que son invocadas cuando se llama a las APIs importadas.

Cada ejecutable puede ser representado como un vector binario de APIs llamado A, donde $A_i = 1$ si y solo si el ejecutable importó la i-ésima función de la API, en cambio, $A_i = 0$ si no la importó.

Este vector de *features* no puede ser utilizado para clasificar adecuadamente ya que las llamadas a la API no son determinantes. Sin embargo, las combinaciones de llamadas a la API pueden predecir si el ejecutable es malware o no de manera más certera.

Tomando como base a SAVE se pueden considerar similitudes entre archivos PE. Los conjuntos de llamadas a la API frecuentes pueden ser tomados como un conjunto de *features*.

Se tomaron las categorizaciones hechas por el sitio de documentación oficial de Microsoft MSDN[9]. Esta categorización está formada por 95 grupos. Se demostró que categorizando de esta manera se lograron mejores resultados.

Como selección de *features* se tomó al *Fisher score* donde se eligieron las llamadas a la API más representativas y se las agregó al conjunto de categorías de MSDN. Como resultado se obtuvo la siguiente lista de categorías más representativas: *File Management*, *Process* and *Thread*, *Console* y *Registry*.

---

[9] https://docs.microsoft.com/en-us/previous-versions//aa383686(v=vs.85)?redirectedfrom=MSDN

Finalmente, ya seleccionadas las categorías y las features, pueden ser enviados a clasificadores como Naïve Bayes, Random Forest o SVM. [16]

Definición 9: *(Fisher score): Se define al score de Fisher como* $Fr = \frac{\sum_{i=1}^{c} n_i(\mu_i - \mu)^2}{\sum_{i=1}^{c} n_i \sigma_i^2}$ *donde* n *es el número de muestras en la clase* i, μ_i *es el promedio de la feature en la clase* i, σ_i *es la desviación estándar de los valores de las features en la clase* i *y* σ *es el promedio de la feature en todo el dataset. El score es máximo cuando* σ_i *es mínimo y la suma de las diferencias entre los promedios de las clases* μ_i *es máximo. Las features con mayor similitud tienen un score más alto.*

3.2.9 Límites del Análisis Estático

Los sistemas actuales que detectan código malicioso están mayormente basados en firmas sintácticas. Es decir, tiene una base de datos con firmas de malware conocido y si el ejecutable en cuestión corresponde con alguna firma, el ejecutable es detectado como *malware*. Esto es a grandes rasgos cómo funciona la detección.

Recientemente se ha demostrado que las técnicas como polimorfismo y metamorfismo son efectivas para evadir este tipo de detección. Esto se debe a que las firmas sintácticas ignoran la semántica de las instrucciones. La premisa de los detectores basados en semántica es que las propiedades semánticas son más difíciles de transformar de una manera automática en comparación con las propiedades sintácticas.

La detección basada en semántica se enfrenta al problema de clasificar si un bloque de código presenta o no un determinado comportamiento no es posible decidirlo de antemano en el caso general. Por otro lado, no es trivial para el creador de *malware* generar código semánticamente equivalente.

Se presenta el caso de las constantes opacas. Es posible modificar el flujo del programa o asignarle la dirección a una variable según el valor que tome una determinada constante. Asignarle valor a una constante es una de las operaciones más sencillas en ensamblador, solo basta con una sola operación. La instrucción puede ser representada con una secuencia de código semánticamente equivalente que sea difícil de analizar estáticamente.

Para ello, se parte del problema *NP-Hard* de la satisfacibilidad booleana (3-SAT) para calcular la constante en cuestión. Encontrar un algoritmo que determine el resultado de la misma implicaría ser capaz de resolver el problema *NP-Hard.* Un analizador estático que intenta determinar exactamente los posibles valores de la constante opaca tiene que resolver la instancia del problema 3-SAT. [22]

Un ejemplo de una constante opaca basado en 3-SAT es el siguiente

```
Boolean $v_1, \dots, v_m, \overline{v_1}, \dots, \overline{v_m}$;
boolean $*V_{11}, *V_{12}, *V_{13}$;
...
boolean $*V_{n1}, *V_{n2}, *V_{n3}$
constant = 1;
for (i = 0; i < n; i++)
    if !($*V_{i1}$) && !($*V_{i2}$) && !($*V_{i3}$)
        constant = 0;
```

Ilustración 11 *Constante opaca basado en el problema 3-SAT.*

Los punteros V_{n1} a V_{n3} hacen referencia a las cláusulas del problema 3-SAT basado en las variables $v_1 \dots v_m$. En este ejemplo solo se tiene un bit de información opaca, pero puede ser extendido para crear constantes arbitrariamente largas.

Definición 10: *(problema 3-SAT): Es un problema de decisión que consiste en encontrar los valores booleanos* $V_{ij} \in \{v_1, \dots, v_m\}$ *y* $v_1, \dots, v_m$ *tales que la fórmula* $\wedge_{i=1}^{m}(V_{i1} \vee V_{i2} \vee V_{i3})$ *sea evaluada como verdadera. Está probado como un problema NP-Hard.*

Con las constantes opacas se obtiene un método para cargar el valor de una constante en memoria sin dejar que un analizar estático sepa su valor. Esto puede ser extendido para aplicar transformaciones que ofusquen el flujo de control, localizaciones de datos y uso de datos por parte del programa,

La idea que está detrás de la ofuscación del flujo de control está en el reemplazo de los saltos condicionales e instrucciones de llamadas a función (*call*) con una secuencia de instrucciones que no alteren el flujo de control. El objetivo es dificultarle al analizador determinar el destino de dicha transferencia de flujo.

En otras palabras, se intenta hacer lo más difícil posible identificar las aristas en un CFG.

Tanto los saltos como los *calls* pueden ser directos o indirectos. En el caso directo, es necesario proveer la dirección destino como una constante. Es posible calcular dicha constante como una constante opaca, guardarla en la pila y con un simple *ret*[10] (para el caso *x86* y *x86_64*) transferir el control a la dirección previamente calculada guardada en el tope de la pila. De esta manera no se revela hacia dónde va el salto. Esta medida hace prácticamente imposible la reconstrucción del CFG.

La ubicación de los datos es por lo general indicada por medio de una constante, una dirección absoluta o un *offset* relativo a un registro particular. Para todos los casos, la tarea del analizador estático puede complicarse si el dato que está siendo accedido está oculto. Esto trae aparejado un problema que es cuando los valores son cargados en los registros pueden ser rastreados. Por ejemplo, cuando una función devuelve un valor y este es usado como argumento para otra función. Esta operación por lo general se realiza por medio de registros y el analizador estático puede identificar esta relación. Para complejizar esta situación, es posible ofuscar la presencia de valores en los registros. Para realizar esta tarea es posible insertar código que copie el contenido de un registro en una posición de memoria ofuscada y luego volver a cargarlo. Para esto, es necesario calcular la posición de memoria temporal por medio de una constante opaca, guardar el contenido del registro y limpiarlo. Luego, cuando sea necesario utilizar los datos, realizar la operación inversa para obtener el valor en el registro. Para este proceso se utilizan posiciones de memoria sin uso en la pila. Luego de aplicar este tipo de ofuscación, el analizador estático identificará dos accesos a memoria sin relacionar.

---

[10] https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-67/index.html

Conclusiones y trabajo futuro

El análisis de malware es un campo interesante que sumado a los nuevos avances en materia de *machine learning* ayuda a los analistas a realizar su labor. Tiene limitaciones que en cierta medida quedan resueltas con el análisis dinámico. Este último quedó a modo de mención en el presente trabajo.

Se deja una introducción y la posibilidad de ser continuado en el futuro mostrando los aspectos generales de las líneas de investigación actuales.

Como próximos pasos, se abordará sobre el análisis dinámico y la combinación entre ambos. El objetivo del próximo trabajo es la propuesta y evaluación de técnicas de análisis junto con *machine learning* para simplificar el laborioso proceso de análisis de malware. Asimismo, evaluar si es posible combinar los aspectos mencionados en este trabajo para proveer una solución que sea demostrable y con aceptables niveles de detección.

Bibliografía

[1] N. Nilsson, Introduction to Machine Learning, Stanford: Stanford University, 1996.

[2] «intermediasoftware.com,» 23 Abril 2021. [En línea]. Available: https://intermediasoftware.com/es/machine-learning/.

[3] «Wikipedia,» 23 Abril 2021. [En línea]. Available: https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico.

[4] D. a. M. W. Wolpert, «Coevolutionary free lunches,» *IEEE Transactions on Evolutionary Computation*, p. 721–735, 2005.

[5] K. R. Bhatele, H. Shrivastava y N. Kumari, «The Role of Artificial Intelligence in Cyber Security,» 2019.

[6] [En línea]. Available: https://www.split.io/glossary/false-positive-rate/. [Último acceso: 7 Julio 2021].

[7] «definicionabc.com,» 2021. [En línea]. Available: https://www.definicionabc.com/tecnologia/seguridad-informatica.php.

[8] C. Manning y H. Schütze, An Introduction to Information Retrieval, Cambridge University Press, 2009.

[9] R. Gandhi, «Towards Data Science,» 5 Mayo 2018. [En línea]. Available: https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c. [Último acceso: 15 Mayo 2021].

[10] R. Khandelwal, «medium.com,» 16 Noviembre 2018. [En línea]. Available: https://medium.datadriveninvestor.com/k-nearest-neighbors-knn-7b4bd0128da7.

[11] S. Raschka y V. Mirjalili, Python Machine Learning, Birmingham: Packt Publishing Ltd., 2017.

[12] V. Gustavsson, «Machine Learning for a Network-based Intrusion Detection System,» 2019.

[13] I. Biju y A. R. Syed, «Performance Comparison of Intrusion Detection Systems and Application of Machine Learning to Snort System,» 2017.

[14] D. Gibert, C. Mateu y J. Planes, «The rise of machine learning for detection and classification of malware: Research developments, trends and challenges,» 2019.

[15] I. Santos, J. Devesa, Y. Penya y P. Bringas, «N-grams-based File Signatures for Malware Detection,» ICEIS, 2009, pp. 317-320.

[16] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi y A. Hamze, «Malware detection based on mining API calls,» 2010.

[17] N. Bhodia, P. Prajapi, F. Di Troia y M. Stamp, «Transfer Learning for Image-Based Malware Classification,» 2019.

[18] J. Kinable y O. Kostakis, «Malware Classification based on Call Graph Clustering,» 2010.

[19] M. Hassen y P. Chan, «Scalable Function Call Graph-based Malware Classification,» 2017.

[20] O. Kostakis, J. Kinable, H. Mahmoudi y K. Mustonen, «Improved Call Graph Comparison Using Simulated Annealing,» 2011.

[21] M. Eskendari y S. Hashemi, «Metamorphic Malware Detection using Control Flow Graph Mining,» 2011.

[22] A. Sung, J. Xu, S. Mukkamala y P. Chavez, «Static Analyzer of vicious executables (save),» 2004.

[23] A. Moser, C. Kruegel y E. Kirda, «Limits of Static Analysis for Malware Detection,» 2007.