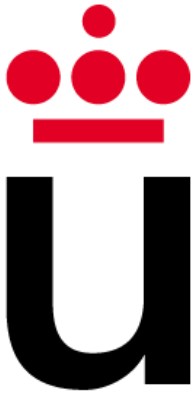


Universidad
Rey Juan Carlos

Clasificación y predicción de vulnerabilidades en ciberseguridad basada en técnicas de Machine Learning

Trabajo de Fin de Grado

Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2016-2017

Sergio Michiels Mangas

Tutor:

Miguel Romance Del Río

Índice general

Capítulo 1

Presentación - Motivación práctica y descripción

En este proyecto se presenta el desarrollo de una aplicación para realizar minería de datos sobre la *National Vulnerability Dabase*, NVD [26], del *National Institute of Standards and Technology*, NIST [25], con el objetivo de hacer una clasificación que sirva para predecir en el futuro la criticidad o ciertas características de nuevas vulnerabilidades. Se utilizarán diversos métodos clásicos de Machine Learning no supervisado, Clustering, y Machine Learning supervisado.

Se escogió este tema porque la ciberseguridad es un tema muy presente actualmente y es uno de los temas que más preocupa a empresas y gobiernos. Se pueden encontrar noticias que hablan de las decenas de miles de ataques al mes que sufren determinadas empresas, o sobre la gran cantidad de dinero que se invierte para protegerse de dichos ataques. En 2015 se movieron más de 60.000 millones de euros en el mundo relacionados con la ciberseguridad y se estima que dicha cifra aumentará hasta los 150.000 millones en 2020 [8]. El daño que puede producir un ciberataque o una fuga de datos no es solo económico, sino que también pueden afectar a la reputación y a la confianza de los clientes.

El Machine Learning es otro tema de actualidad que puede ayudar a prevenir ciberataques. Las empresas cada vez tienen más datos, más información, y necesitan sistemas con inteligencias artificiales más avanzadas, capaces de tratar y procesar esos datos. Si analizamos la información de bases de datos como la NVD o la CVE [3], *Common Vulnerabilities and Exposures*, se podría intentar extraer patrones o averiguar las características claves de ciertos ciberataques o vulnerabilidades con el objetivo de prevenir nuevos ataques,

cerrándoles las puertas a los atacantes.

Se decidió escoger la base de datos del NIST por los siguientes motivos:

- La CVE [3], mencionada anteriormente, es una lista de información registrada sobre vulnerabilidades de seguridad creada por el MITRE [24]. A primera vista, la base de datos del NIST puede parecer un espejo de la CVE, ya que también es una base de datos de vulnerabilidades y utilizan la misma nomenclatura que la CVE, pero la NVD contiene mucha más información, análisis de vulnerabilidades y un mejor motor de búsqueda. La NVD está sincronizada con la CVE de tal forma que cualquier actualización de está última aparece inmediatamente en la NVD.

- El volumen de datos que posee. En el momento de escribir esta memoria, la base de datos contiene aproximadamente unas 80000 vulnerabilidades. Una cantidad adecuada para realizar diferentes métodos de aprendizaje automático sin que el tiempo de cómputo en los equipos disponibles sea un inconveniente.

- La estructura de la información de las vulnerabilidades. El NIST tiene disponibles diferentes esquemas para la información, utilizando diferentes versiones del CVSS [4], vectores de scores, etc. Además toda la información almacenada es pública y no hay ningún tipo de restricción de uso sobre la misma.

- Los mecanismos dispuestos por el NIST para la extracción de datos. La base de datos ofrece una serie de "*Data feeds*" en ficheros .xml para que quien lo desee pueda utilizar la información disponible en la base de datos.

- La información de la base de datos no se borra. Las vulnerabilidades permanecen en la base de datos y se marcan como *rejected*, pero permanecen allí.

Escogí realizar este proyecto por mi interés en las técnicas de aprendizaje automático, la minería de datos y la seguridad informática. Considero que son campos muy interesantes y que tienen una gran demanda en el panorama profesional actual, y quería indagar e investigar más acerca de ellos. Además, el set de datos de las vulnerabilidades proporcionado por el NIST es bastante interesante, ya que podemos extraer de él información como patrones o clasificaciones en las vulnerabilidades de los últimos 17 años, y es importante conocer toda la información que se pueda sobre vulnerabilidades, ya

que con esa información podemos intentar predecir si un sistema está en una fase crítica, si es objeto de ciberataques frecuente o periódicamente, o bien si los parches para dichas vulnerabilidades están funcionando o abriéndoles nuevas puertas a los atacantes. Quizás no permita predecir vulnerabilidades tan importantes como *Heartbleed* [14], pero toda la información es bienvenida.

El proyecto se estructura de la siguiente manera:

- Primero se hará una descripción de la información disponible en la base de datos, además de un pequeño análisis de la temporalidad de las vulnerabilidades.
- A continuación se detallarán brevemente los fundamentos matemáticos de los algoritmos que se han implementado en el proyecto.
- Se continuará con un resumen de las herramientas informáticas utilizadas. Lenguaje, IDEs, tecnologías, etc.
- Después se continuará con un resumen del modelo de datos creado para almacenar la información de la NVD.
- Acto seguido, se entrará más en detalle de los algoritmos programados, su funcionamiento, y la estimación de sus parámetros.
- Luego se dedicará un apartado a las conclusiones y al trabajo futuro.
- Finalmente se incluirá un breve manual de instalación y uso.

Capítulo 2

Descripción de la base de datos

En este capítulo se va a hacer una introducción a la base de datos de vulnerabilidades del NIST, su volumen de datos, cómo proporciona esos datos y un pequeño estudio estadístico de la temporalidad de las vulnerabilidades de ciertos sistemas en concreto.

El *National Institute of Standards and Technology*, NIST, es una agencia de la Administración de Tecnología del Departamento de Comercio de los Estados Unidos. Se encarga de promover la innovación y la competencia industrial en Estados Unidos mediante avances en metrología y tecnología, con la misión de refinar la ciencia de la medición creando una ingeniería precisa para mejorar la estabilidad económica y la calidad de vida. Se puede acceder a su web a través del enlace que aparece en la bibliografía de este trabajo [25]. La NVD, accesible también desde el enlace listado al final de este trabajo [26], es una base de datos de vulnerabilidades que incluye información como métricas de impacto, descripciones de producto, relaciones entre vulnerabilidades a través del software afectado, etc. La NVD se distingue de la CVE en que aporta información sobre la cuantificación del riesgo de las vulnerabilidades, calculando scores utilizando ecuaciones complejas que incluyen variables como la presencia de parches, la complejidad en el acceso o similares.

En la figura 2.1 podemos observar el número de vulnerabilidades presentes en la base de datos y su evolución en el tiempo. En el momento de escribir esta memoria hay un total de 80000 vulnerabilidades almacenadas, una cifra, como se dijo antes, suficiente para realizar un pequeño estudio de data mining y que el tiempo de cómputo no sea un impedimento.

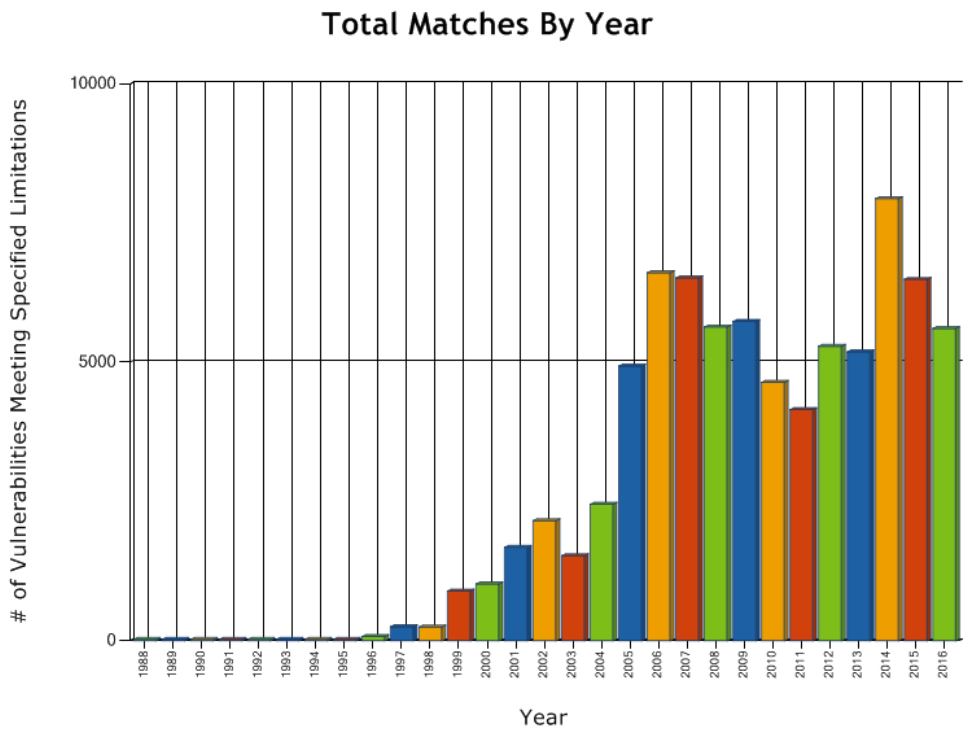


Figura 2.1: Evolución en el tiempo de las vulnerabilidades de la NVD (Fuente: [25])

Figura 2.2: Contenido de los ficheros .meta

Para la extracción de la información, el NIST proporciona una serie de herramientas en forma de *feeds* para poder descargar la información de la base de datos, organizando las vulnerabilidades en paquetes por años, para facilitar el trabajo con la información en local. Para simplificar las cosas, el NIST agrupa en el diccionario de 2002 todas las vulnerabilidades de la base de datos desde 1988 hasta entonces.

Además de dichos feeds, el NIST proporciona una serie de archivos .meta como el de la figura 2.2 con los siguientes datos de los diccionarios:

- La fecha de la última modificación del diccionario. Día, mes y año además de la hora completa.
- Tamaño del diccionario.
- Tamaño del diccionario comprimido en un .zip.
- Tamaño del diccionario comprimido en un .gz.
- El sha256 del diccionario.

Estos archivos .meta se utilizarán para actualizar la información local de la aplicación y llevar un seguimiento de su estado.

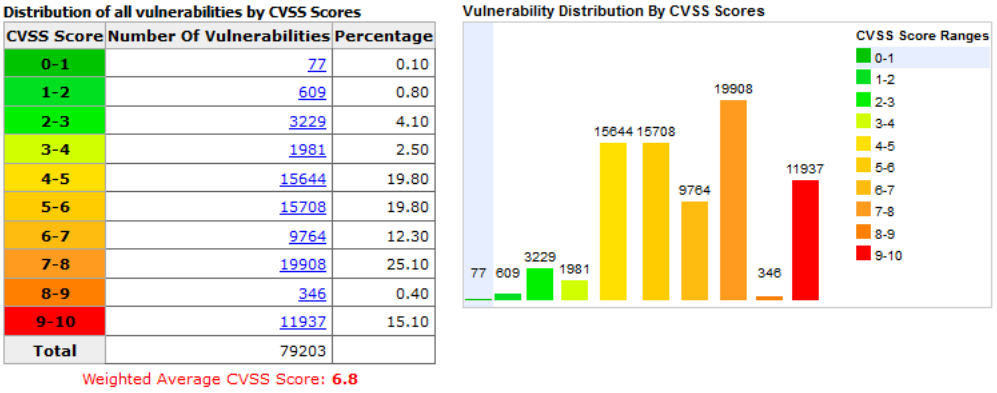
Para cada una de las vulnerabilidades de la base de datos, se nos presenta una página similar a la de las figuras 2.5 y 2.6. En ellas podemos observar los datos de la vulnerabilidad en diferentes sistemas de scoring (CVSS2 y CVSS3) si están disponibles. Para la aplicación desarrollada en este TFG se ha decidido usar el sistema CVSS2, ya que la versión 3 no está disponible en todas las vulnerabilidades y si mezcláramos diferentes sistemas las métricas

no coincidirían.

Para cada vulnerabilidad podemos encontrar tres scores que la representan: *Base score*, *Impact score* y *Exploitability score*. Además, cada vulnerabilidad posee un vector de características que determinan conceptos como si afecta a la confidencialidad o el vector de acceso. El vector de CVSS3 posee 8 elementos, mientras que el de CVSS2 posee 6. En la tabla 2.1 puede verse una comparativa.

Comparativa entre vectores	CVSS2	CVSS3
Access/Attack vector	Local, Adjacent network, Network	Local, Adjacent network, Network, Physical
Access/Attack complexity	Low, Medium, High	Low, High
Authentication required	None, Single instance, Multiple instances	X
Privileges required	X	None, Low, High
User interaction	X	None, Required
Scope	X	Unchanged, Changed
Confidentiality impact	None, Partial, Complete	None, Low, High
Integrity impact	None, Partial, Complete	None, Low, High
Availability impact	None, Partial, Complete	None, Low, High

Tabla 2.1: Variaciones entre los vectores de CVSS2 y CVSS3



Distribution of all vulnerabilities by CVSS Scores

CVSS Score	Number Of Vulnerabilities	Percentage
0-1	77	0.10
1-2	609	0.80
2-3	3229	4.10
3-4	1981	2.50
4-5	15644	19.80
5-6	15708	19.80
6-7	9764	12.30
7-8	19908	25.10
8-9	346	0.40
9-10	11937	15.10
Total	79203	

Weighted Average CVSS Score: **6.8**

Figura 2.3: Distribución de las vulnerabilidades de la CVE. Fuente: [5]

En la parte inferior de la página se detalla quién ha informado de esta vulnerabilidad, si ha sido el fabricante u otra entidad, enlaces a la página del desarrollador, además de una lista del software y las versiones afectadas por la vulnerabilidad. También se pueden ver los cambios o actualizaciones que ha sufrido la información de la vulnerabilidad. Todo esta información puede verse en la figura 2.6.

También puede ser interesante analizar el conjunto de las vulnerabilidades de la base de datos. Como se puede observar en las figuras 2.3 y 2.4, la mayor parte de las vulnerabilidades tienen un score entre 7 y 8, siendo la score media de todas las vulnerabilidades 6.8. Así mismo, se puede ver cómo la mayoría de las vulnerabilidades son aquellas que permiten al atacante ejecutar código arbitrario y aquellas que permiten ataques de denegación de servicio.

Para el estudio previo de la base de datos decidió darse un pequeño paso más, escribiendo una serie de scripts en Python que sirvieran para hacer gráficas con la temporalidad de las vulnerabilidades del sistema de que se desee. Así podría obtenerse otro punto de vista de la información de la NVD, ver la evolución de estos sistemas en lo que a su seguridad se refiere, así como intentar detectar patrones o averiguar cada cuánto se descubre una vulnerabilidad crítica en dicho sistema. A continuación se muestran algunos ejemplos.

La NVD no proporciona figuras como la 2.7, 2.8 o 2.9, y podría ser interesante analizar estos gráficos para sacar otro tipo de conclusiones.

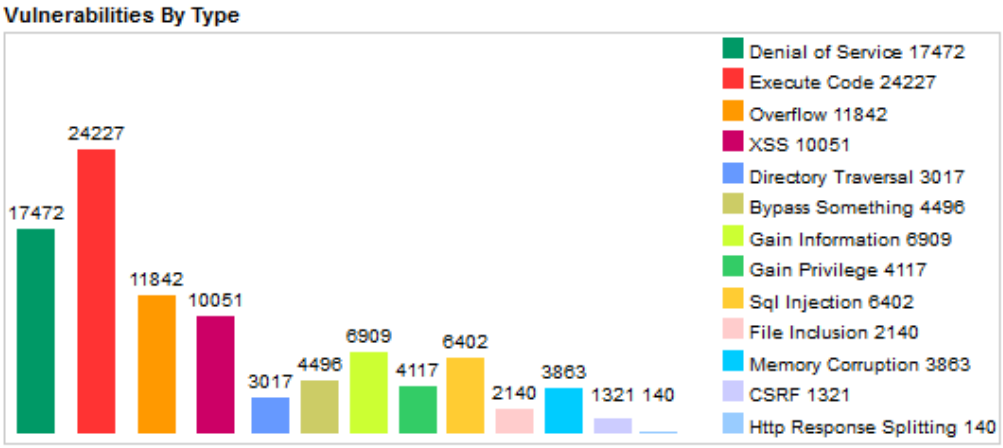


Figura 2.4: Desglose de las vulnerabilidades de la CVE por tipo. Fuente: [34]

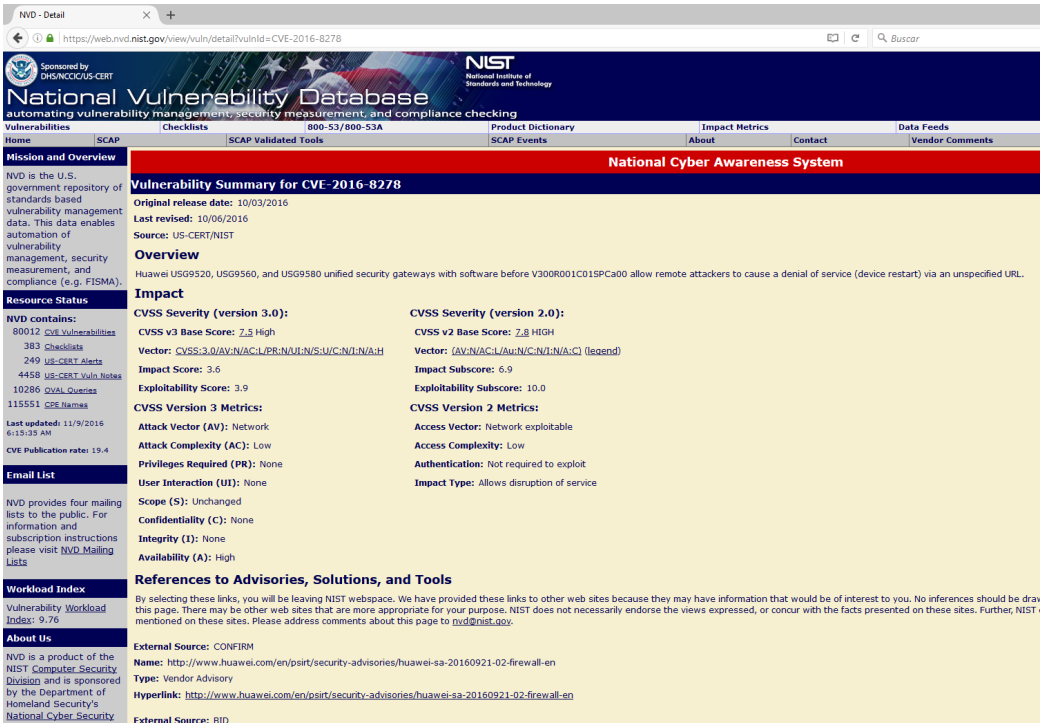


Figura 2.5: Entrada de una vulnerabilidad (parte 1)

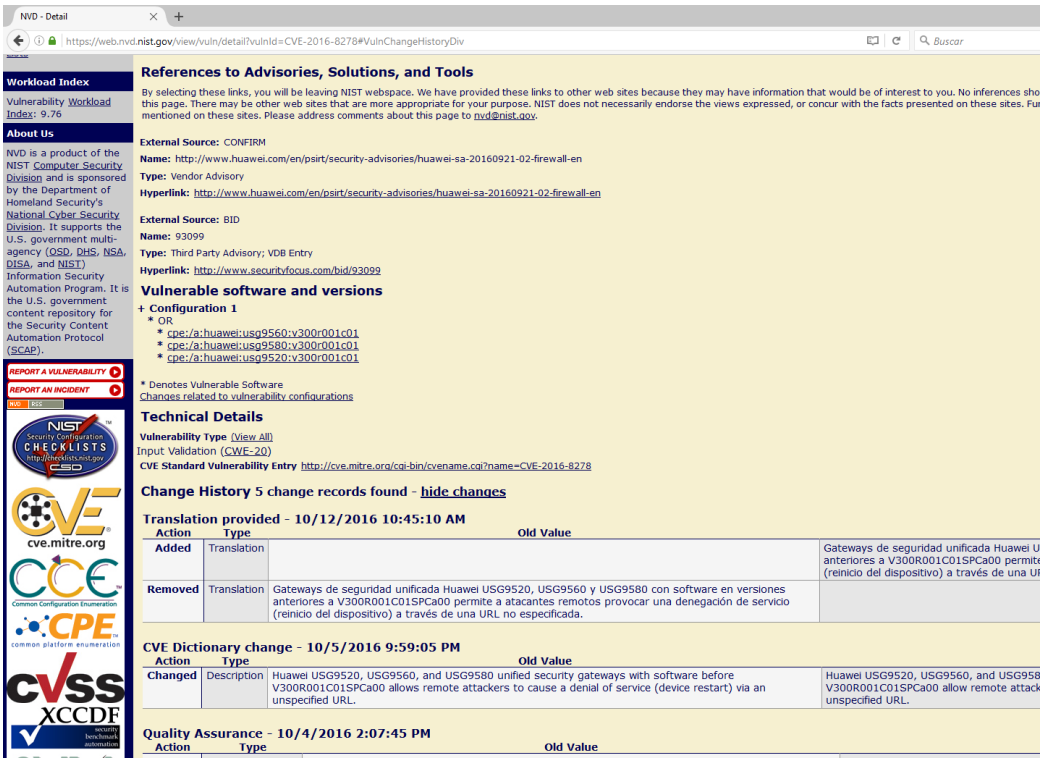


Figura 2.6: Entrada de una vulnerabilidad (parte 2)

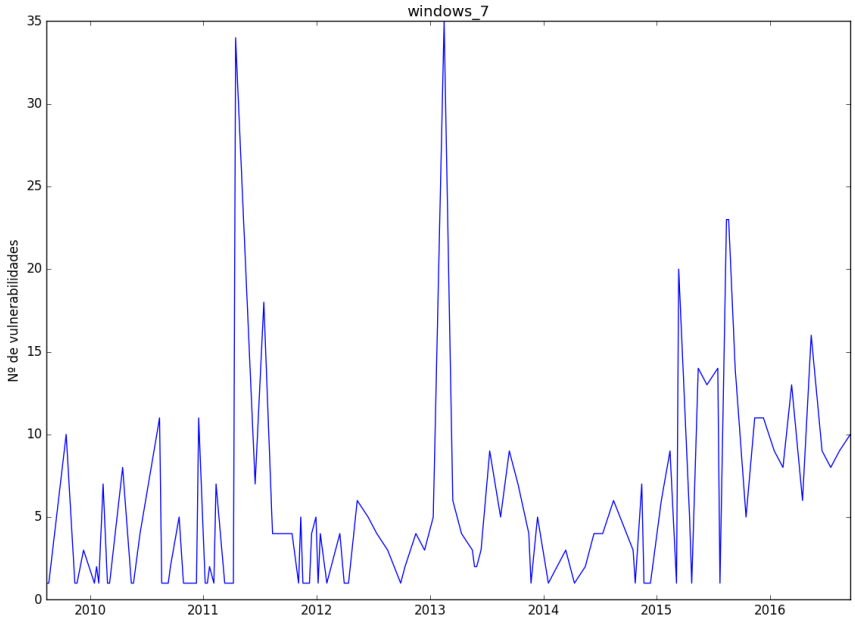


Figura 2.7: Evolución de vulnerabilidades (no acumuladas) en Windows 7

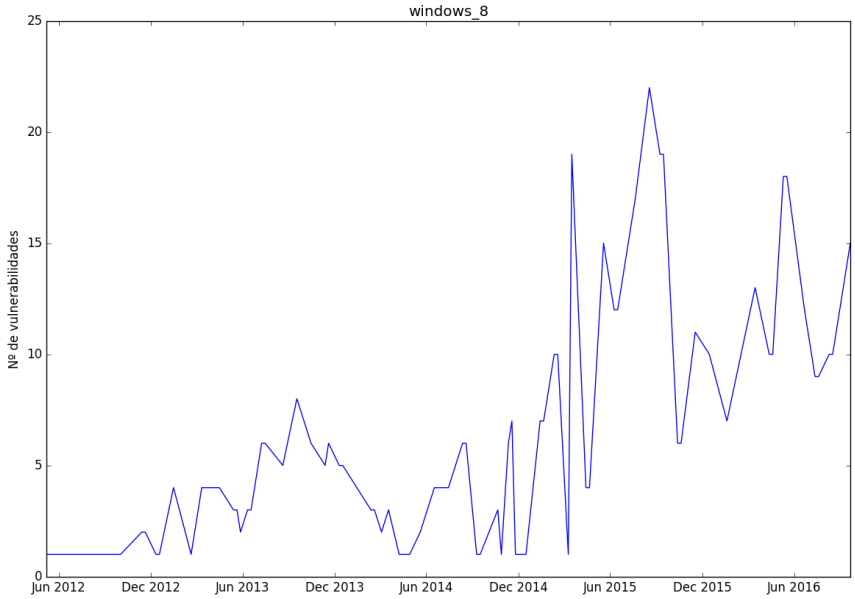


Figura 2.8: Evolución de vulnerabilidades (no acumuladas) en Windows 8

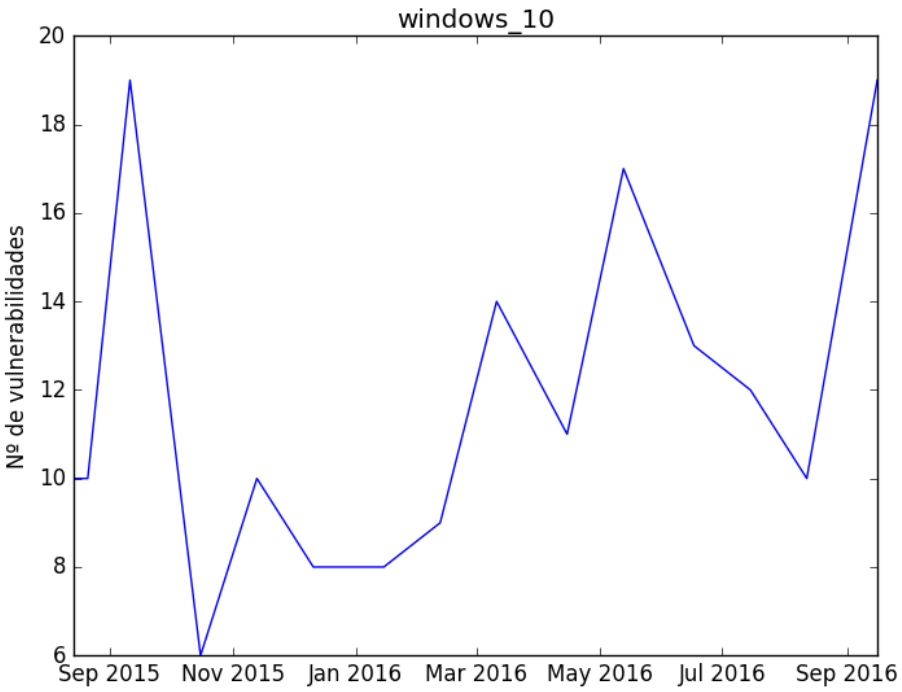


Figura 2.9: Evolución de vulnerabilidades (no acumuladas) en Windows 10

Por ejemplo, en la figura 2.7, podemos ver claramente dos picos de vulnerabilidades en abril de 2011 y febrero de 2013. Si analizamos en detalle esas fechas, podemos averiguar qué es lo que pasó. En abril de 2011 se destaparon una serie de vulnerabilidades relacionadas con el kernel de Windows, en el que un driver permitía que un atacante local pudiera ejecutar código arbitrario con privilegios superiores, debido a una mala gestión de objetos erróneos por parte de dicho driver del kernel, *win32k.sys*. Afectó a varias versiones de Windows, desde XP SP3 a W7 pasando por Vista y varias versiones de Windows Server. La vulnerabilidad fue anunciada por Microsoft junto con un gran paquete de vulnerabilidades relacionadas con el mismo problema. De ahí el pico de esa fecha [23].

En febrero de 2013 se produjo un escenario similar, en el que Microsoft publicó doce boletines que revelaban 57 vulnerabilidades. De esas vulnerabilidades, 30 tuvieron que ver con el mismo driver de hace dos años y la vulnerabilidad que permitía un ascenso de privilegios a nivel local [20].

En la figura 2.8 de Windows 8 se pueden apreciar por ejemplo dos picos de vulnerabilidades relativamente próximos aunque no tan altos como los de su sistema predecesor. Hablamos de los picos de marzo y agosto de 2015. Si analizamos detenidamente el pico de marzo, encontramos una serie de vulnerabilidades de diferentes temas:

- Vulnerabilidades relacionadas con un *Adobe Font Driver*: El cual permitía a un atacante ejecutar código de manera remota si la víctima visualizaba archivos creados de manera específica o una web determinada. Permitían al atacante controlar totalmente el sistema afectado [21].

- Numerosas versiones de Windows Server, Vista, W7 y W8 permitían a un atacante remoto ejecutar código arbitrario vía un fichero de *Windows Journal* modificado.

El pico de agosto se debe principalmente a una vulnerabilidad bastante crítica relacionada con *Windows Adobe Type Manager Library*, que permitía a un atacante remoto ejecutar código arbitrario en un sistema y tomar su control total cuando la víctima abriera un fichero malicioso directamente o al visitar alguna página web que tuviera incrustado contenido con fuentes *OpenType*. Esta vulnerabilidad afectaba a Windows Vista, W7 y W8 además de varias versiones de Windows Server, y Microsoft tuvo que lanzar un par-

che de urgencia [9].

En la figura 2.9 de Windows 10 podemos observar algunos picos en septiembre de 2015 y recientemente, en septiembre de 2016. No son picos tan pronunciados como los de sus predecesores, pero puede ser interesante estudiarlos.

El pico de septiembre de 2015 se debe principalmente a una serie de vulnerabilidades relacionadas con *Windows Journal* y *Windows Media Center*. Ambas vulnerabilidades permitían a atacantes remotos ejecutar código arbitrario y tomar control del sistema, al abrir la víctima un fichero .jnt modificado (en el caso de *Windows Journal*) o a través de un enlace mcl (en el caso de *Windows Media center*). Eran vulnerabilidades muy críticas que recibieron un parche en apenas semanas.

El otro pico de septiembre de 2016 se debe también a vulnerabilidades de alta criticidad que permiten que un atacante ejecute código arbitrario sobre el sistema:

- Con un archivo pdf modificado a través del lector de PDFs de W8.1, Windows Server 2012 y W10.
- A través de SMBv1 de Vista, 7, 8.1 o 10, utilizando paquetes modificados.
- A través de los drivers del Kernel de Vista, 7, 8.1 o 10, utilizando una aplicación modificada.
- Utilizando un documento modificado y con el GDI de W10.

De este análisis podemos extraer la conclusión de que las vulnerabilidades que más afectan a Microsoft son aquellas que permiten ejecutar código arbitrario por parte del atacante. Los picos siguen estando presentes a o largo del tiempo, pero podemos apreciar un ligero descenso en lo que llevamos de 2016, como muestra la figura 2.10.

Con estos tipos de gráficas se podría hacer un análisis más exhaustivo de la NVD y extraer información descriptiva de las vulnerabilidades. Sin embargo, como se puede apreciar en las gráficas, es difícil encontrar un patrón en ellas, por lo que de cara a un estudio predictivo como el que queremos hacer, parece que este estudio no aporta mucha información. Por tanto, en los siguientes capítulos, vamos a proceder a describir los algoritmos de Machine Learning

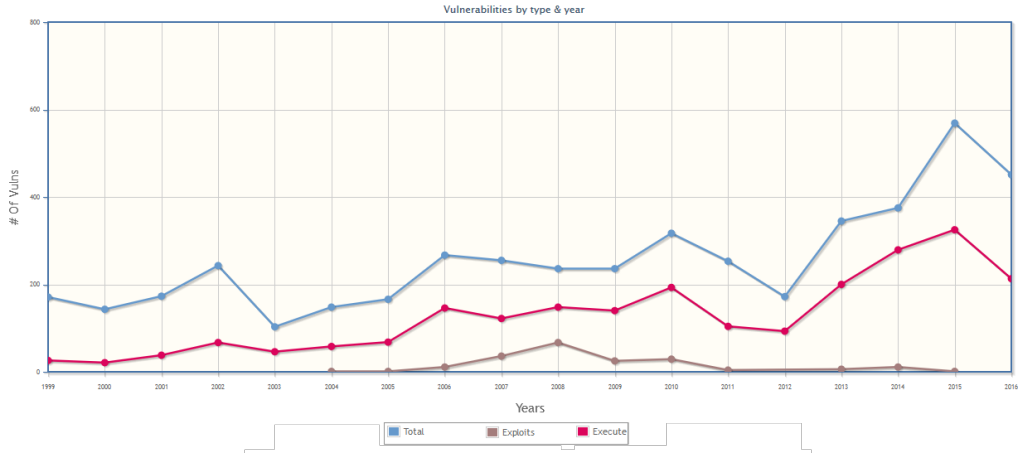


Figura 2.10: Evolución de vulnerabilidades en productos Microsoft. Fuente: [22]

que se han implementado en este proyecto, las herramientas utilizadas, y sus resultados.

Capítulo 3

Descripción de los algoritmos y fundamentos matemáticos

En este capítulo se van a describir los algoritmos de Machine Learning empleados en la aplicación así como su base matemática. Se comenzará por los algoritmos de clusterización para terminar con los algoritmos de validación y PCA. Son los siguientes:

- Algoritmos de clusterización: K-means, clustering jerárquico y DBSCAN.
- Algoritmos de clasificación: Knn y Máquinas de Soporte Vectorial.
- Otros algoritmos: Análisis de Componentes Principales.

3.1. Algoritmos de clusterización

3.1.1. K-means

K-means es un algoritmo heurístico que realiza una clasificación y agrupamiento sobre un set de datos en un número determinado de grupos ya predefinido. El algoritmo necesita minimizar una función que representa la similitud o la distancia entre dos observaciones [19].

El algoritmo es el siguiente:

Algoritmo 3.1.1 *Dado un conjunto de observaciones* $(x_1, x_2, \ldots, x_n)$*, donde cada observación es un vector real de dimensión d, K-means construye una partición de las observaciones en k conjuntos* $(k \leq n)$ *a fin de minimizar una función de coste de cada grupo.*

Existen diferentes implementaciones del algoritmo, cada una con ciertos pasos diferentes. En nuestro caso se ha escogido la versión estándar del algoritmo, también conocida como "*Algoritmo de Lloyd*" [19]:

Algoritmo 3.1.2 *Dado un conjunto inicial de centroides (generado aleatoriamente), el algoritmo continúa un número determinado de iteraciones repitiendo estos dos pasos:*

- *Paso 1: asignación*
  A cada observación del set de datos se le asigna el centroide más cercano o más similar a él.

- *Paso 2: actualización*
  Recalcular cada uno de los centroides como la media de todas las observaciones que fueron asignadas a cada uno de ellos.

El algoritmo se considera que ha convergido cuando las asignaciones ya no cambian.

Para la generación de los centroides hay diferentes aproximaciones que han presentado los expertos. Ciertas variedades del algoritmo como "*K-means con partición aleatoria*" asignan una partición a cada una de las observaciones al comenzar el algoritmo, o la "*inicialización de Forgy*", que fue la elegida a la hora de implementar el algoritmo para este proyecto, que consiste en hacer una selección inicial aleatoria de k observaciones del set de datos y utilizarlas como centroides iniciales. A partir de ahí los centroides se irán recalculando y ya no tienen por qué representar una observación real [19].

Otro aspecto del algoritmo que puede cambiar es el concepto de distancia, ya que hablamos de centroide más cercano, o minimizar una suma de cuadrados o coste. Existen varias maneras de definir la distancia o la diferencia entre dos observaciones del conjunto de datos. Dado que contamos con vectores de dimensión d en un espacio euclídeo, podemos utilizar cualquiera de las normas ya definidas. En muchos otros estudios de Machine Learning se recomienda el uso de otros conceptos para medir la similitud entre observaciones, como el Coeficiente de Correlación de Pearson o la definición del coseno en $\mathbb{R}^n$. En este caso se ha decidido utilizar le definición del coseno (3.2) con alguna ligera variación para las distancias entre características cualitativas, ya que la librería NumPy proporciona los métodos necesarios en álgebra lineal para realizar esos cálculos fácilmente.

3.1.2. Clustering jerárquico

La clusterización jerárquica (HCA, *Hierarchical cluster analysis* en inglés), es un método de clusterización que busca realizar una jerarquía de grupos. Su particularidad es que no necesita de antemano un número de clases en las que clasificar los datos. A partir de un dendograma se elige una línea de corte y se forman los clusters sin la intervención previa de alguien.

Algoritmo 3.1.3 *Las estrategias del clustering jerárquico caen en dos tipos:*

- **Aglomerativo***: Es un acercamiento ascendente. Cada observación comienza en su propio cluster y dichos clusters se van juntando mientras se sube en la jerarquía.*
- **Divisivo***: Es un acercamiento descendente. Se comienza con un único grupo que contiene todos los datos y se realizan divisiones mientras se baja en la jerarquía.*

Como se dijo antes, los resultados del algoritmo se suelen representar con un dendograma. Un dendograma es una representación gráfica de los datos en forma de árbol, organizando los datos en categorías que se van dividiendo en otras. A partir de ahí se determina una línea horizontal que corte las ramas del árbol. Dichos cortes determinarán los clusters finales, que serán los árboles hijos que queden.

Para realizar el agrupamiento de clusters o la división de los mismos, se necesita definir una métrica para el conjunto de datos y un criterio de enlace

para determinar las distancias entre clusters a partir de los datos que los forman. Algunas de las distancias más usuales son la euclídea (3.1) o la definición del coseno (3.2). Existen también numerosos criterios de enlace entre los que elegir: tomar como referencia los puntos que más lejos estén entre sí, los más cercanos, utilizar los centroides de los cluster, etc. En el capítulo 6 se enunciarán aquellos criterios que soporta la aplicación.

Normalmente, los algoritmos de clustering jerárquico no son óptimos para set de datos muy grandes, ya que tienen una complejidad de $O(n^2 \log(n))$ o incluso $O(2^n)$ si son divisivos, además del espacio en memoria que necesitan, ya que suelen computar matrices de distancias muy grandes. Más adelante se comprobará si esto es cierto o no.

$$d_E(u, v) = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2} \tag{3.1}$$

$$\cos(u, v) = \frac{\langle u, v \rangle}{|u||w|} \tag{3.2}$$

3.1.3. DBSCAN

El agrupamiento espacial basado en densidad de aplicaciones con ruido (Density-based spatial clustering of applications with noise, DBSCAN, en inglés) [7], es un algoritmo de clusterización propuesto por Martin Ester, Hans-Peter Kriegel, Jörg Sander y Xiaowei Xu en 1996. Se basa en la densidad porque encuentra clusters comenzando por una estimación de la distribución de densidad de los datos. Es uno de los algoritmos más utilizados en los estudios de data mining. Fue premiado en 2014 en la KDD (una de las conferencias líderes en la minería de datos) con el premio a la prueba de tiempo.

La base de DBSCAN son los siguientes conceptos:

- Un punto p se considera un *punto núcleo* si al menos Min_Pts puntos están a una distancia ε o menor de él (incluyéndose él mismo). Dichos puntos se dicen que son *directamente alcanzables* desde p.
- Un punto q se considera alcanzable desde p si existe una secuencia de puntos $p_1, p_2, \ldots, p_n$ donde $p_1 = p$ y $p_n = q$ tal que cada punto p_{i+1}

es directamente alcanzable desde p_i. Es decir, todos los puntos de la secuencia son núcleos (con la posible excepción de q).

- Todo punto no alcanzable desde ningún otro punto se considera *ruido*.

Entonces, si p es un núcleo, formará un cluster con todos aquellos puntos (núcleos o no) que son alcanzables desde él. Cada cluster contendrá al menos un punto núcleo; puntos no núcleo pueden formar parte del mismo y de hecho serán sus "bordes", ya que no se pueden alcanzar más puntos desde ellos.

De aquí podemos observar que la relación de "ser alcanzable desde" no es simétrica, ya que por definición no hay puntos que sean alcanzables desde un punto no núcleo. Es decir, un punto no núcleo puede ser alcanzable, pero ninguno otro será alcanzable desde él. Por tanto se define una relación de *conexidad* para definir la extensión de los clusters.

- Dos puntos p y q se dicen densamente conectados si existe un tercer punto o, tal que p y q son directamente alcanzables desde o. Es una relación simétrica.

- Todos los puntos de un cluster estarán densamente conectados entre sí.

- Si un punto p es directamente alcanzable desde otro cualquier punto q del cluster, entonces también formará parte del cluster.

En lo relativo a la complejidad, queda determinada principalmente por la implementación de cómo obtener la vecindad de un punto. Ya que esta búsqueda se tiene que ejecutar una vez por punto, en el mejor de los casos el algoritmo puede alcanzar una complejidad de $O(n \log(n))$. En el peor de los casos su complejidad es de $O(n^2)$, resultado no del todo aceptable.

El principal problema que nos presentará este algoritmo es la elección de los parámetros ε y Min_Pts. Se entrará más en detalle en este problema en el capítulo 6.

3.2. Algoritmos de clasificación

3.2.1. Knn

K nearest neighbors, Knn [7] [13] [10], es un algoritmo de Machine Learning supervisado que realiza una clasificación sobre un conjunto de datos habiendo sido previamente entrenado con otro set de datos ya clasificado. Al ser un algoritmo de aprendizaje supervisado, necesitamos un "*experto*" que realice el entrenamiento del algoritmo y que clasifique los datos del set de entrenamiento para que sirvan como base. El algoritmo es el siguiente:

Algoritmo 3.2.1 *Sean los ejemplos de entrenamiento vectores en un espacio característico de dimensión p.*

$$x_i = (x_{1i}, x_{2i}, \ldots, x_{pi}) \in X. \tag{3.3}$$

Un punto en el espacio es asignado a la clase $c \in C$ si esta es la clase más frecuente entre los k ejemplos de entrenamiento más cercanos o similares a él.

$$\hat{f}(x) \leftarrow \arg\max_{c \in C} \sum_{i=1}^{k} \delta_v^{f(x_i)}(c, f(x_i)), \tag{3.4}$$

donde $\delta_v^{f(x_i)}$ es la Delta de Kronecker [6].

En nuestro caso, no contamos con ningún experto que nos pueda ayudar a realizar una clasificación de las vulnerabilidades, pero podemos aprovechar los resultados de K-means o de cualquier otro algoritmo de clustering como base de este algoritmo.

3.2.2. SVM

Las máquinas de vectores de soporte [7] [13] [10], máquinas de soporte vectorial (*Support Vector Machines*, SVMs), son un conjunto de algoritmos de Machine Learning supervisado. Dados una serie de ejemplos ya clasificados o etiquetados, el algoritmo de entrenamiento de una SVM genera un modelo que asignará nuevos ejemplos a las categorías correspondientes.

El modelo de una SVM es una representación de los datos como puntos en el espacio, de tal manera que las observaciones de diferentes categorías queden separadas por un "hueco" lo más grande posible. Los nuevos ejemplos que lleguen se representan en dicho espacio y se asignarán a una clase u a otra dependiendo de la región del espacio en la que caigan.

La manera más simple de realizar la separación entre dos conjuntos del espacio es mediante una línea recta, un plano recto o un hiperplano n-dimensional. Sin embargo, no se suelen presentar estos casos ideales, y el algoritmo tendrá que tratar con curvas o delimitadores más complejos o casos en los que los datos no pueden ser completamente separados. Para solucionar estos inconvenientes de las máquinas de aprendizaje lineal, se utilizan los "*kernel*". La representación mediante estas funciones *kernel* proyectan la información a un espacio de dimensión mayor a la original, lo cual aumenta la capacidad de computación. Más adelante en el capítulo 6 se determinarán qué kernels son compatibles con la aplicación y qué parámetros necesitan.

3.3. Otros algoritmos

3.3.1. PCA - Análisis de componentes principales

El Análisis de Componentes Principales, ACP, (Principal Component Analysis, PCA, en inglés), es un algoritmo de Machine Learning para reducir la dimensionalidad de un set de datos. PCA busca la proyección en un hiperplano de dimensión inferior al espacio actual según la cual los datos queden mejor representados. Para ello, se basa en la descomposición en valores singulares de una matriz que contiene información de los datos tras haber sido centrados respecto a su media, pero en este trabajo no se va a entrar en detalles matemáticos sobre este algoritmo.

El beneficio que aporta PCA es que retiene aquellas características del set de datos que contribuyen más a su varianza, y dependiendo de el límite que se especifique para dicha retención de la variabilidad, seleccionará las características más apropiadas de los datos para que la información no se distorsione al reducirla al número de dimensiones deseado.

PCA se puede llevar a cabo a partir de una descomposición de la matriz de covarianza de los datos o utilizando una descomposición en valores singulares. Para este proyecto se ha decidido utilizar este último método, ya que es más común en los estudios de Machine Learning. Toda la demostración del teorema y sus consecuencias se encuentran en mi memoria del Trabajo de Fin de Grado de Matemáticas [19].

Capítulo 4

Descripción informática de las herramientas

Los programas, lenguajes y herramientas utilizadas para el desarrollo de esta aplicación son los siguientes:

- **Lenguaje: Python** [30]
  Python es un lenguaje con tipado dinámico, más flexible que otros como Java y permite mezclar orientación a objetos y programación imperativa. Es algo lento a la hora ejecución, y tenemos el inconveniente del GIL (el Global Interpreter Lock es un mutex que previene que varios threads de Python ejecuten bytecode a la vez [11]) por la parte de multi-threading, pero su sintaxis es algo más simple y posee características como la indentación del código que aportan legibilidad. Otro punto a favor de Python es la gran cantidad de librerías que existen para realizar todo tipo de tareas, tanto en la librería estándar de Python como en librerías hechas por otros usuarios.
  Como alternativas a Python se consideraron Java o R, pero al final se desecharon por las siguientes razones:

  - Pese a la cantidad de paquetes disponibles en R para realizar data mining, hay elementos del lenguaje como el manejo de las estructuras de datos que no parecen del todo cómodos. Además, la intención era hacer todo el proyecto en un mismo lenguaje, y la interfaz gráfica suponía un obstáculo importante en R.
  - Java es un lenguaje popular en todo el mundo, y principalmente el lenguaje que más se utiliza durante el grado, pero tras haber aprendido otros lenguajes como Python por cuenta propia, elementos como la mayor legibilidad, la comunidad, el sistema de

indentación, y el gran número de librerías disponibles ponen a Python por delante de Java.

Para esta aplicación se ha empleado Python 3.5.2, que es la versión más actual del lenguaje en el momento de comenzar este proyecto. Pese al legado de Python2 y ciertas librerías que no han sido porteadas a Python3, este último es el futuro del lenguaje, y ha recibido numerosas actualizaciones desde 2010, cuando se creó. Se decidió usar esta versión ya que para una persona que no ha tenido que adaptarse al cambio y empezó aprendiendo Python3, no hay mucho más donde elegir. Hoy en día casi todo es compatible con Python3, y programar en esta versión garantiza estabilidad y soporte durante más tiempo, ya que el fin al soporte de Python2 llegará tarde o temprano.

- **Distribución: Anaconda** [1]
  Python es un lenguaje muy cómodo, pero para disponer de las librerías necesarias para trabajar con datos y realizar gráficas hay que ir visitando los diferentes repositorios de Python para descargar e instalar las herramientas correspondientes. Anaconda es una distribución "*freemium*"de Python, compatible con Python2 y Python3 que instala los paquetes más importantes para el tratamiento de grandes volúmenes de datos, cálculos complejos y análisis de predicciones. A la hora de instalarse, permite que se especifique Anaconda como una versión de Python más, para que la gran mayoría de los IDEs que se suelen utilizar lo reconozcan como una versión más del lenguaje. Se incluyen paquetes como NumPy, SciPy, Numba, Matplotlib... Para este proyecto se ha utilizado la versión de Anaconda compatible con Python 3.5.2. De las 4 licencias disponibles de Anaconda una es gratuita (la utilizada), mientras que las otras tres son de pago. Las dos licencias de mayor coste incluyen un paquete llamado "*Accelerate*"que incluye librerías para pre-compilar código en Python y guardarlo y ejecutarlo desde la GPU, para utilizar CUDA y acelerar scripts escritos en Python apenas modificando el código ya existente. No se ha utilizado está funcionalidad debido al alto coste económico. Para este proyecto se ha empleado la versión 4.1.1 de Anaconda, la más reciente al comienzo de este proyecto.

- **IDE: PyCharm Community Edition** [28]
  PyCharm es un entorno de desarrollo integrado para Python. Es multiplataforma, proporciona análisis de código, depuración gráfica e integración con control de versiones entre otras muchas utilidades. Está desarrollado por JetBrains y dispone de dos versiones con sus respectivas licencias: la versión "*Professional*", es de pago y bajo licencia

propietaria; y la utilizada en este proyecto, la versión "*Community*", gratuita y bajo licencia Apache. La licencia Apache es una licencia de software libre permisiva creada por la Apache Software Foundation, ASF [2]. Se ha empleado la versión de PyCharm 2016.2.3.

- **GitHub** [12]
  Github es una plataforma de desarrollo colaborativo para alojar proyectos utilizando Git como control de versiones. Este proyecto está alojado en mi GitHub personal, público para todo el mundo. Se ha elegido esta plataforma para llevar un histórico de los cambios realizados en el proyecto según iba avanzando su desarrollo y para facilitar el transporte del código, ya que he trabajado en diferentes ordenadores.

- **Qt y PyQt** [29] [31]
  Qt es un framework multiplataforma orientado a objetos utilizado para desarrollar aplicaciones que utilicen interfaz gráfica de usuario. Es desarrollada como software libre y de código abierto, y se utiliza en KDE, un entorno de escritorio para sistemas Linux. De forma nativa está escrito en C++, pero posee bindings para ser utilizado en otros lenguajes. En este proyecto se emplea PyQt, un binding para Python hecho por *Riverbank Computing*, y disponible bajo GPL y otras licencias comerciales. Para este proyecto se han empleado Qt 4.8.7 y PyQt 4.11.4, versiones ya incluidas en Anaconda.

- **QtDesigner** [32]
  QtDesigner es una herramienta de Qt para el diseño y desarrollo de interfaces gráficas para aplicaciones utilizando componentes Qt. Permite componer widgets y diálogos, probarlos y utilizar diferentes estilos, de la manera conocida como "what you see is what you get" (WYSIWYG), lo que ves es lo que obtienes, es decir, permitiendo ver el resultado final durante la construcción de la interfaz. Utilizando el mecanismo *slots and signals* se pueden conectar los widgets de la interfaz a los métodos que se deseen, para que se ejecuten cuando se realiza cierta acción. Está disponible bajo licencia LGPL. Se ha empleado la versión 4.8.7.

Capítulo 5

Descripción del modelo de datos

A continuación se va a describir el modelo escogido para almacenar, representar y operar con las vulnerabilidades de la base de datos del NIST.

5.1. Modelo de vulnerabilidad

Para cada una de las vulnerabilidades almacenadas en la base de datos tenemos una gran diversidad de datos, tal y como indicamos en el capítulo 2. Toda esta información aparece representada en diferentes esquemas de datos en los *feeds* del NIST. Además, algunas de ellas poseían sus datos en diferentes sistemas de scoring de vulnerabilidades debido a su antiguedad. Para definir un tipo de datos, fijamos un sistema de scoring y extrajimos la información más importante. Cada vulnerabilidad se plasma en un objeto que posee los siguientes atributos:

- Identificador CVE [3]
  - Un identificador para la vulnerabilidad.
- Fecha de publicación y fecha de última modificación
  - Fechas importantes relacionadas con la vulnerabilidad, aunque son meramente informativas, ya que para actualizar los datos locales tenemos en cuenta las fechas de los diccionarios en los que se guardan dichas vulnerabilidades. Lo explicaremos en la siguiente sección.

- *Base score*, *Impact subscore* y *Exploitability subscore*
  - Scores importantes para la vulnerabilidad. Todas toman valores entre 0 y 10. La *Base score* es la puntuación "principal" de la vulnerabilidad.
- *Severity*
  - Es un rating adicional a los scores numéricos de la vulnerabilidad. Dependiendo del *Base score* de la vulnerabilidad puede tomar los valores *HIGH*, *MEDIUM* o *LOW*.
- Vector de información acerca de la vulnerabilidad
  - Tipo de acceso necesario para explotarla
  - Complejidad del acceso
  - Autenticación necesaria para explotarla
  - Impacto a la confidencialidad
  - Impacto a la integridad
  - Impacto a la disponibilidad
- Descripción de la vulnerabilidad
- Software y versiones afectadas por la vulnerabilidad

Para los algoritmos de Machine Learning se decidió utilizar el vector ya existente como vector representativo de la vulnerabilidad. Sin embargo, esto presentaba un pequeño problema, ya que no incluía toda la información relevante de la misma. Faltaban los scores y/o la severidad, ya que como hemos podido apreciar, la información que aportan es la misma. Por tanto, se decidió incluir también los tres scores dentro del vector de la vulnerabilidad, para así operar con vectores de dimensión 9.

Más tarde, durante la ejecución de las primeras pruebas de los algoritmos de Machine Learning se nos presentó otro problema. Los vectores presentan características numéricas así como características cualitativas, siendo estas últimas incompatibles con los algoritmos ya programados en este proyecto. Para solucionar este problema se decidió hacer un mapeo entre los valores que dichas características podían tomar y valores numéricos con los que ya se podría operar sin problemas al ejecutar los algoritmos sobre el set de datos. Es una solución fácil y aceptable, pero al resolver el problema nos presenta uno nuevo: las diferencias entre estas características no eran equitativas,

es decir, la distancia entre los valores numéricos de *bajo* - *medio* no era la misma que entre los de *bajo* - *alto*. En cierto modo se podría pensar que lo correcto es que las distancias no fueran equitativas, pero al no conocer en detalle los criterios que utiliza el NIST para establecer sus scores así como si entre ellos hay algunos más importantes que otros a la hora de calcular el score principal, en caso de duda se escoge distancia genérica 1 cuando no se tiene la misma característica y distancia cero en caso contrario. Para resolver este imprevisto, se realizaron las modificaciones pertintentes a los métodos del módulo *distances*, que incluye una variante de la definición del coseno para adaptar los valores de los vectores a este criterio.

El NIST avisa de que en ocasiones reciben o encuentran vulnerabilidades de las que no poseen toda su información o detalles, bien porque el desarrollador del software reconoce la vulnerabilidad pero no aporta todos sus detalles o similares. En esta situación, el NIST posee un equipo de analistas que realizan un análisis para dar unos scores teniendo en cuenta el peor caso posible. Si un desarrollador no proporciona ningún detalle sobre cierta vulnerabilidad, esta se puntúa con el score más alto, 10.0. [27]

Las vulnerabilidades se almacenan en unos diccionarios que comentaremos en la siguiente sección.

5.2. Modelo de diccionario de vulnerabilidades

Cuando se extraen las vulnerabilidades de la base de datos del NIST, se van almacenando en unos objetos que representan diccionarios de vulnerabilidades que tienen los siguientes atributos:

- Año del diccionario
    - Tanto los feeds de datos que proporciona el NIST como los ficheros relacionados con el estado de actualización de los mismos vienen dados por años. El NIST no informa de qué vulnerabilidades han sido actualizadas de manera individual, sólo avisa de cuándo fue actualizado un determinado diccionario. Por tanto, cuando el usuario quiera actualizar la información local que posee, se actualizan diccionarios enteros, no vulnerabilidades concretas. Por esta razón, las vulnerabilidades son agrupadas en diccionarios por años, para preservar la estructura de la información y facilitar las futuras actualizaciones que se realicen sobre los datos.
- Diccionario
    - Es el diccionario en sí en el que se almacenan las vulnerabilidades. Las claves son los identificadores CVE de las mismas, ya que son únicos.
- Fecha de la última modificación
    - Es la fecha que sirve de referencia cuando se quiere actualizar la información del diccionario. Se consulta al NIST la fecha en la que se produjo la última actualización del diccionario. Si esta fecha es más reciente que la última fecha de modificación local del diccionario, quiere decir que los datos de alguna de las vulnerabilidades del mismo han cambiado, pero no se nos indica exactamente cuál o cuáles. Por tanto, actualizamos toda la información del mismo y sobrescribimos todos los datos. Finalmente se marca la fecha de la última modificación como la del día correspondiente.

El principal problema que se presentaba aquí era el proceso de actualización de los datos. Ya que el NIST no nos proporciona la información exacta de qué vulnerabilidad ha sido actualizada, se nos obligaba a mirarlas una a una e ir comparando o a actualizar todo el diccionario del año correspondiente. Se optó por esto último. Es una tarea algo más costosa computacionalmente

hablando, pero para este volumen de datos, sabiendo que ningún año posee más de 10000 vulnerabilidades almacenadas, no supone un incremento significativo en el tiempo de ejecución.

A la hora de ejecutar cualquier algoritmo de aprendizaje automático, se vuelca el contenido de los diccionarios que se vayan a utilizar en una lista y se operará con ella.

Capítulo 6

Descripción informática de los componentes

A continuación, se describirán los módulos empleados en la aplicación del proyecto. Son los siguientes:

- Módulo data_types: Contiene los tipos de datos de vulnerabilidades y diccionarios, así como los métodos para extraer la información del NIST y procesarla.
- Módulo de clustering: Contiene los módulos con los algoritmos de clusterización. Son:
  - K-means: Clusterización con número de clusters predefinidos.
  - Método del codo para K-means: Estimación de K para el algoritmo K-means.
  - Hierarchical clustering: Clusterización jerárquica aglomerativa.
  - DBSCAN: Clusterización basada en densidades y con detección de ruido.
- Módulo de clasificación supervisada: Contiene los módulos con los algoritmos de clasificación. Son:
  - Knn: Clasificación supervisada que se basa en los vecinos más cercanos a una observación dada.
  - SVM: Clasificación supervisada que intentar establecer un "large margin classifier" para separar las clases de los datos.
- Módulo interfaz gráfica: Contiene los módulos necesarios para las interfaces gráficas de las aplicaciones de entrenamiento y predicción/clasificación.

- Otros módulos:
    - PCA: Un algoritmo de reducción de la dimensionalidad. Para comprimir información o facilitar la computación.
    - Distances: Contiene funciones de distancias.
    - loadAndSave: Contiene métodos de carga y guardado de datos.

6.1. Módulo data_types

Este módulo contiene los tipos de datos de las vulnerabilidades, el software afectado, y los diccionarios en los que almacenar la información. Además, posee los métodos para extraer la información de la base de datos de vulnerabilidades del NIST y actualizar los ficheros locales. Esta última parte se ejecuta en el caso de que el usuario indique que desea descargar la información más reciente en la aplicación, o bien cuando al cargar una serie de ficheros locales alguno de ellos esté dañado.

El procedimiento para descargar o actualizar cualquiera de los diccionarios consiste en conectarse a la web del NIST y descargar los ficheros .zip con las vulnerabilidades allí almacenadas (desde 2002 hasta 2016) para luego descomprimirlos y realizar un análisis a los ficheros .xml extraídos. Para cada uno de esos ficheros, iremos extrayendo la información de todas y cada una de las vulnerabilidades, almacenándolas en diccionarios por años, para mantener la estructura de la información y mantener un mejor control sobre cuándo las vulnerabilidades de cierto año están desactualizadas. Una vez analizadas todas ellas, se procede a borrar tanto el fichero .zip como el .xml. Se marca el diccionario recién creado como actualizado y se guarda en disco. Cada una de las vulnerabilidades se guarda en un objeto que representa el modelo expuesto en el anterior apartado.

6.2. Módulos de clusterización

6.2.1. Módulo K-means

K-means [7] [10] [13] es un algoritmo de Machine Learning no supervisado que realiza una clusterización sobre un dataset en un número determinado de clusters.Tras la presentación del algoritmo en la sección 3.1.1, procedemos a detallar el módulo escrito en Python que lo ejecuta.

Este módulo permite ejecutar el algoritmo K-means sobre un determinado set de datos que recibe como lista. El módulo contiene numerosas funciones, pero la que ejecuta el algoritmo en sí es `kmeans(datalist, times, k)`. El método recibe el propio set de datos, el número de veces que se quieren ejecutar los dos pasos de asignación y actualización mencionados anteriormente y el número de clusters en los que se quiere dividir el set de datos.

También se dispone de otro método:

`run_kmeans(datalist, iterations, times, k)`

Este método recibe el propio set de datos, el número de iteraciones que se quiere correr K-means (para asegurarnos obtener un resultado lo más correcto posible), el número de veces que se quieren ejecutar los dos pasos de asignación y actualización mencionados anteriormente en cada ejecución del algoritmo y el número de clusters en los que se quiere dividir el set de datos. En general, se suelen ejecutar entre 100 y 1000 iteraciones de K-means para intentar asegurarnos de no caer en un resultado incorrecto, habiendo entre 5 y 10 fases de asignación-actualización cada vez que se ejecuta. De todas formas, se ha mantenido todo con parámetros y variables generales para el usuario final pueda especificar el número exacto de iteraciones que desea, poniendo por defecto 100 ejecuciones, 6 fases de asignación-actualización, y división en 4 clusters.

El principal problema que aquí se presentaba era la estructura del modelo de los datos extraídos del NIST, ya que las vulnerabilidades tenían características cualitativas (severidad, riesgo de integridad, confidencialidad...), y como bien se ha indicado anteriormente, K-means requiere de vectores reales de dimensión d. Se barajaron diferentes soluciones, como incluir características binarias para cada uno de los posibles valores de las características cualitativas. Sin embargo, esta idea se desechó ya que generaba un modelo de datos con vectores numéricos de dimensión 30, lo que aumentaría considerablemente el tiempo de cómputo y dificultaría la función de otros métodos

como PCA, cuyo objetivo es la reducción de la dimensionalidad y el intentar así conseguir mejorar los resultados de otros algoritmos. Finalmente, como se comenta en el apartado de la descripción del dataset, la solución escogida fue asignar una serie de valores numéricos a dichas características cualitativas. En vez de incorporar una característica binaria para cada posible valor, incluimos varios valores enteros para todos los valores posibles que pueda tomar la característica cualitativa.

K-means (4 grupos) sin PCA	Tiempo	Porcentaje de aciertos
Entrenamiento	56 minutos	X
Knn(k=4)	27 mins	99,967 %
SVM lineal	0,822 segs	99,995 %
SVM rbf kernel ($\gamma = 0{,}125$)	1,33 segs	99,991 %

Tabla 6.1: Resultados obtenidos con K-means sin PCA

K-means (4 grupos) con PCA	Tiempo	Porcentaje de aciertos
Entrenamiento	54 minutos	X
Knn(k=4)	25 mins	99,885 %
SVM lineal	1,54 segs	99,992 %
SVM rbf kernel ($\gamma = 0{,}125$)	1,67 segs	99,993 %

Tabla 6.2: Resultados obtenidos con K-meann con PCA

Con este algoritmo, un ordenador con una CPU Intel i7 6700K @4.0Ghz tardó aproximadamente 55 minutos en realizar una clasificación con los valores por defecto, obteniendo los resultados de las tablas 6.1 y 6.2.

Para la validación, se define un porcentaje (por defecto 80 %) y se divide el set de datos ya entrenado en dos listas dependiendo del porcentaje. Una de las listas se emplea como set de entrenamiento para el algoritmo de aprendizaje supervisado, ya que la información está etiquetada gracias a K-means. La otra lista se clasifica en los tests finales utilizando el algoritmo supervisado. El objetivo es que el algoritmo supervisado prediga la misma clasificación que definió la clusterización. Así nos aseguramos de que la clasificación que hemos obtenido es buena, ya que habiendo entrenado un algoritmo supervisado solo con parte de los clusters podemos cerciorarnos de que cierta observación está “cerca” de sus vecinos del cluster.

Se realizaron estas pruebas 10 veces, definiendo las listas de entrenamiento y test de manera aleatoria en cada una de las diez pruebas. En las tablas 6.1 y 6.2 se incluyen el tiempo medio de ejecución de cada uno de los tests así como el porcentaje medio de aciertos.

6.2.2. Módulo elbow method K-means

Una de las principales cuestiones que presenta el método de K-means a la hora de ejecutarlo es la elección del número de clusters en los que dividir el conjunto de datos. Realmente no existe una regla que funcione en el 100 % de los casos y sirva para elegir sin lugar a dudas un número de clusters correcto, pero tenemos numerosos test o checks que sirven para calcular un número de clusters que podría ser el adecuado. El llamado "*elbow method*" (método del codo) es el que se ha utilizado en este proyecto.

Algoritmo 6.2.1 *El razonamiento tras este método es que uno debe elegir un número de clusters de tal manera que, si añadiéramos uno más, no se aporte un incremento beneficioso significativo al modelo. De una manera más precisa: si ejecutáramos el algoritmo varias veces variando el número de clusters en los que dividir el dataset, las primeras iteraciones presentarán una distorsión bastante importante, pero según vayamos incrementando el número de clusters la distorsión descenderá, y llegará un momento en el que se estabilizará, ya que estamos añadiendo clusters que no representan un gran cambio en la clasificación que ya se ha establecido con menos clusters.*

Si representamos gráficamente esta relación de la distorsión con el número de clusters, la gráfica resultante presenta un ángulo en un punto. Ese punto es el momento en el que añadir más clusters ya no aporta más distorsión, de ahí el nombre de codo, por el ángulo. Ese punto en concreto puede ser un número adecuado de clusters para nuestro dataset, puesto que más clusters que esos no aportarán mucho más beneficio al modelo. Muchas veces este codo no es fácilmente visible puesto que la variación en la distorsión no es tan pronunciada, pero aun así este método sirve para hacerse una idea de por dónde debemos atacar el problema de la elección de k.

Para este dataset, se probaron clusterizaciones entre uno y ocho clusters, para obtener el resultado de la figura 6.1.

En este caso, podríamos decir que dicho codo está presente cuando el dataset se divide en tres o cuatro clusters, por lo que esos podrían ser valores adecuados para k en el algoritmo K-means. Por tanto, en la implementación del algoritmo, si no se aporta un parámetro k al método que inicia K-means, este por defecto tomará el valor 4.

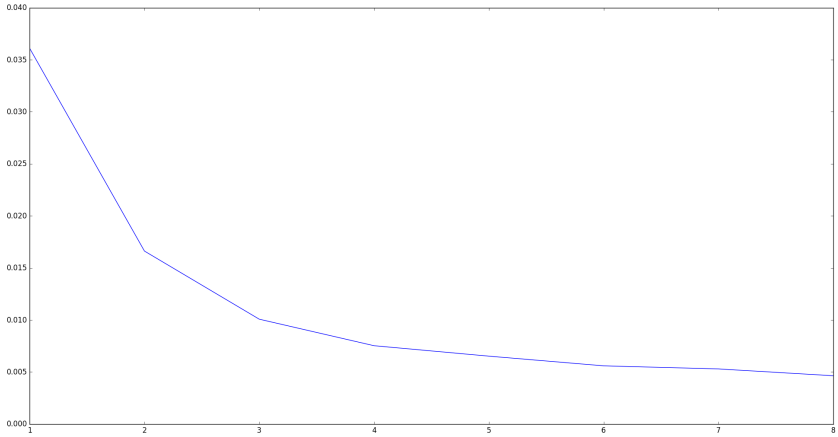


Figura 6.1: Método del codo para k=1...8.

El principal problema que presenta este test es el gran coste computacional que conlleva, ya que tenemos que correr K-means muchas veces con cada vez más clusters, lo que conlleva a un mayor número de operaciones, cálculos de distancias, costes y distorsiones, y por tanto tarda en ejecutarse mucho más tiempo. Para este test, un ordenador con una CPU Intel i7 6700K @4.0Ghz tardó aproximadamente 4 horas y media. Este resultado, pese ser un tiempo relativamente pequeño para la tarea a realizar, fue uno de los motivos que me puso a investigar acerca de CUDA en Python o el multi-threading y multi-processing. La librería *Accelerate* ofrecía una solución interesante y desde luego rápida, pero no pudo llevarse a cabo al no poder costearse el precio de la licencia de Anaconda correspondiente. Finalmente no se incluyó en este módulo nada de programación multihilo, ya que el método del codo no es necesario ejecutarlo cada poco tiempo, pero sí se incluyó en futuros módulos del proyecto.

6.2.3. Módulo Hierarchical clustering

El clustering jerárquico [10] [13] es un método de análisis de grupos que busca construir una jerarquía de grupos. Tras la presentación del algoritmo en la sección 3.1.2, describimos el funcionamiento del módulo.

En este módulo se realiza concretamente una clusterización aglomerativa utilizando la distancia euclídea (3.1). Para ello nos aprovechamos de la implementación que nos proporciona el paquete *SciPy* de Anaconda para ejecutar el algoritmo. El método que ejecuta el algoritmo es

```
run_hierarchical (datalist, criteria)
```

es decir, recibe el propio set de datos para entrenarse y el criterio para decidir la unión de clusters, en inglés, *linkage criteria*. Existe otro método en el módulo que recibe los mismos parámetros que `run_hierarchical()` pero que realiza un plot de parte del set de datos, sin embargo, no se ha utilizado dicho método en la aplicación final debido a limitaciones técnicas de la interfaz gráfica. En la sección de la interfaz se darán más detalles.

Como se comentó antes, existe un gran número de criterios de enlace, incluso se podría definir uno nuevo específico si la situación lo requeriera. En este caso debemos restringirnos a la implementación de Anaconda, la cual permite utilizar los criterios más comunes. Por defecto es el criterio "*ward*', pero el algoritmo soporta los siguientes criterios:

- *single*: Se le conoce como el *Nearest Point Algorithm*. Se eligen aquellos puntos de los clusters que más cerca están entre sí.
- *complete*: Se le conoce como el *Farthest Point Algorithm*. Se eligen aquellos puntos de los clusters que más lejos están entre sí.
- *average*: Se le conoce como el *UPGMA algorithm*. Se calculan las distancias entre todos los puntos y se dividen entre la cardinalidad de los clusters.
- *weighted*: Se le conoce como el *WPGMA algorithm*.
- *ward*: Se le conoce como el *incremental algorithm*.

Pese a que nuestro set de datos (80000 vulnerabilidades) no es tan grande comparado con los estudios habituales de big data y data mining, durante el entrenamiento se pudo percibir el problema que se presentó en el apartado 3.1.2: los algoritmos de clustering jerárquico no son los más idóneos para

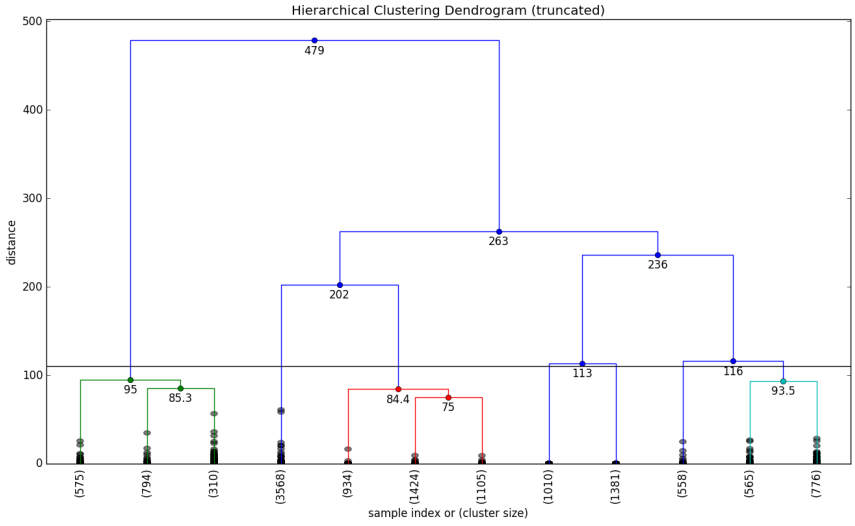


Figura 6.2: Dendograma con 13000 vulnerabilidades de 2009-2016

trabajar con sets grandes.

El primer problema que se presentó fue el espacio en memoria. Para este tipo de agrupamientos con la implementación que aporta Anaconda, son necesarias operaciones con matrices y la creación de una matriz *nxn* con las distancias entre puntos. Si se realizan las cuentas, utilizando como tipo de datos *float32*, para una matriz de 80000x80000 se necesitarían aproximadamente 24GB de memoria. El ordenador en el que se ha desarrollado este Trabajo de Fin de Grado sólo dispone de 8GB de memoria RAM, por lo que no se puede realizar un entrenamiento con todo el set de datos, siendo el máximo permitido 43000 vulnerabilidades, ya que si no Python levanta una excepción de memoria.

Jerárquico aglomerativo sin PCA	Tiempo	Porcentaje de aciertos
Entrenamiento	157 minutos	X
Knn(k=4)	25 mins	99,86 %
SVM lineal	0,51 segs	99,846 %
SVM rbf kernel ($\gamma = 0{,}125$)	0,61 segs	99,969 %

Tabla 6.3: Resultados obtenidos con clustering jerárquico aglomerativo sin PCA

Jerárquico aglomerativo con PCA	Tiempo	Porcentaje de aciertos
Entrenamiento	151 minutos	X
Knn(k=4)	25 mins	99,783 %
SVM lineal	1,11 segs	99,97 %
SVM rbf kernel ($\gamma = 0{,}125$)	0,89 segs	99,967 %

Tabla 6.4: Resultados obtenidos con clustering jerárquico aglomerativo con PCA

Incluso el tiempo de ejecución del algoritmo es demasiado alto aun habiendo tratado antes los datos con PCA y reduciendo su dimensionalidad de 9 a 5 para agilizar la computación. Para realizar un clustering jerárquico aglomerativo de las vulnerabilidades de entre 2009 y 2016 (aprox. 43000 vulnerabilidades), un ordenador con una CPU Intel i7 6700K @4.0Ghz tardó aproximadamente 2 horas y media. Además, para realizar el plot del dendograma y elegir el punto a partir del cual tomar los clusters, también se presentan problemas en lo que a recursividad se refiere, siendo 13000 puntos el máximo permitido para incluir en un dendograma en este ordenador, incluso aun habiendo aumentado antes el límite de recursividad permitido por Python. Por tanto, se corre el riesgo de que el plot que se hace para presentar el dendograma y guardar los clusters no sea representativo del árbol general, ya que incluye un 30 % de la información (Figura 6.2). Este fue uno de los motivos junto a la limitación de la interfaz gráfica que hicieron que se eliminara el plot del método final que se utiliza.

Además, los resultados obtenidos son parecidos, incluso algo peores, que los obtenidos con K-means, como se puede apreciar en las tablas 6.3 y 6.4. Por lo que quizás el clustering jerárquico no sea el método más adecuado para nuestro set de datos.

6.2.4. Módulo DBSCAN

DBSCAN es otro de los algoritmos de clustering más utilizados. Presenta la misma ventaja que el clustering jerárquico, que no necesita el número de clusters de antemano, y otra mejora más, detección de ruido. Tras presentar el algoritmo en la sección 3.1.3, vamos a detallar cómo calcular los parámetros necesarios para DBSCAN y posteriormente describiremos el módulo que lo ejecuta.

Tal y como se dijo antes, DBSCAN necesita dos parámetros para clusterizar el conjunto de datos:

- ε, que determina el radio de búsqueda de vecinos para cada punto del set de datos. Dependiendo de la distancia que se escoja para el algoritmo se debe tomar un valor adecuado. Más adelante veremos las pruebas que hay que hacer para ello.
- *Min_pts*, que determina el número de puntos vecinos que debe tener un punto para ser considerado núcleo y formar un cluster. Igualmente, más adelante veremos las pruebas necesarias para elegir un valor adecuado para el conjunto de datos.

Algoritmo 6.2.2 *Una vez determinados dichos parámetros, el algoritmo comienza por un punto no visitado escogido aleatoriamente. Se marca el punto como visitado y se calcula su "vecindad" basándonos en el ε especificado. Si contiene el número de puntos especificados, se le cataloga como punto núcleo y se inicia un cluster. De lo contrario, se le etiqueta como ruido.*

Si se inicia un cluster, se procede a expandirlo. Se crea una lista con los vecinos del núcleo del cluster. Para cada elemento de la lista se calcula su vecindad, si es del tamaño suficiente, se agregan los vecinos del punto a dicha lista y el punto en cuestión se incluye en el cluster si no es miembro de ningún cluster ya, dándolo por analizado.

Una vez vaciada dicha lista de vecinos, se pasa al siguiente punto del set de datos y se realiza el mismo procedimiento.

El método que ejecuta el algoritmo es

```
def run_dbscan(datalist, eps, min_pts)
```

el cual recibe como parámetros la lista con el set de vulnerabilidades de entrenamiento, y los dos parámetros que se deben haber estimado previamente.

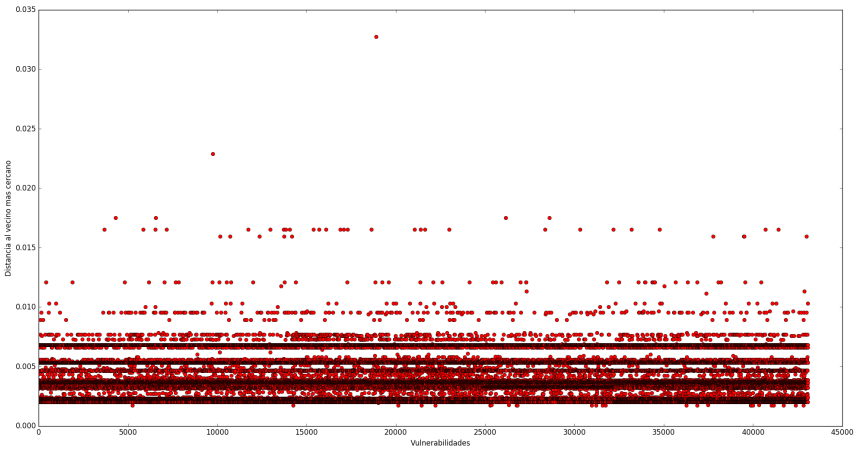


Figura 6.3: Cálculo del vecino más cercano de cada punto para obtener ε

En caso de no aportarse ε o Min_Pts, tomarán los valores por defecto que se especifican más adelante en esta sección.

Uno de los problemas que posee el algoritmo es la complejidad en memoria. Ciertas implementaciones del algoritmo (como la facilitada en Anaconda) necesitan de una matriz de distancias para ejecutar el algoritmo. Esta matriz $n \times n$ de floats puede ocupar mucho espacio en memoria, por lo que o bien se reimplementa el algoritmo cambiando el almacenamiento de los datos que genera o bien se dispone de un equipo de mejores capacidades. En este caso, el ordenador en el que se desarrolló el Trabajo de Fin de Grado cuenta sólo con 8 GB de RAM, por lo que solo se puede entrenar el algoritmo con la mitad del set de datos. La solución a este problema es muy similar a la del problema con el clustering jerárquico. O bien dotar al sistema de más memoria para operar con más datos o reimplementar el algoritmo entero, haciendo uso de bases de datos y métodos que no necesitan de una matriz de distancias para obtener las vecindades de los puntos, pero como se comentó antes, no es el caso de la implementación que se encuentra en los paquetes de Anaconda.

Para la elección de los parámetros ε y Min_Pts, se realizan las siguientes pruebas (están incluídas en el módulo en Python):

- Para ε: Para cada punto del set de datos calculamos la distancia que hay entre cada punto y su vecino más cercano, para después hacer un plot con los resultados. Utilizando la distancia del coseno, se obtiene la

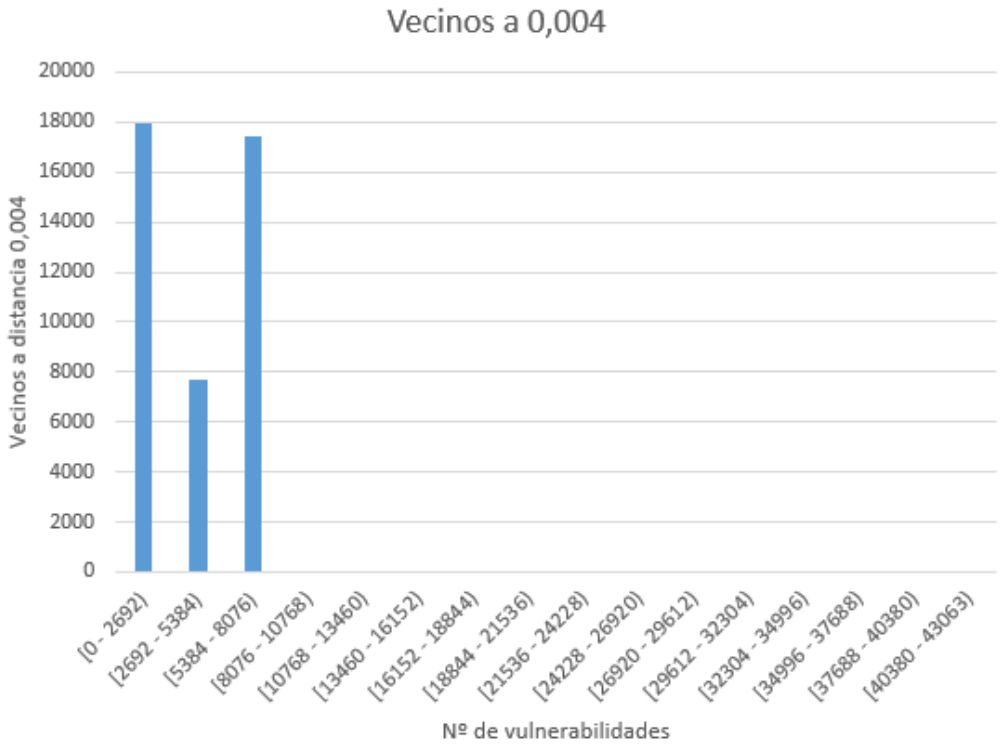


Figura 6.4: Tamaño de la vecindad de cada punto dado ε

DBSCAN sin PCA	Tiempo	Porcentaje de aciertos
Entrenamiento	264 minutos	X
Knn(k=4)	23 mins	100 %
SVM lineal	0,25 segs	100 %
SVM rbf kernel ($\gamma = 0{,}125$)	0,3 segs	100 %

Tabla 6.5: Resultados obtenidos con DBSCAN sin PCA

DBSCAN con PCA	Tiempo	Porcentaje de aciertos
Entrenamiento	262 minutos	X
Knn(k=4)	22 mins	100 %
SVM lineal	0,92 segs	100 %
SVM rbf kernel ($\gamma = 0{,}125$)	0,88 segs	100 %

Tabla 6.6: Resultados obtenidos con DBSCAN con PCA

figura 6.3. Como se puede apreciar, la mayoría de los puntos tienen su vecino más cercano aproximadamente a 0.004 unidades. Por tanto ese valor para epsilon puede ser razonable.

- Para *Min_Pts*: Una vez definido ε, calculamos para cada punto el tamaño de su vecindad, con el objetivo de hacer un gráfico de barras similar al de la figura 6.4. Como se puede observar, aproximadamente un 40 % de las observaciones poseen sus vecinos en torno al primer intervalo, de 0 a 2692 puntos en su vecindad (para el ε dado). A partir de ahí el intervalo siguiente tiene números menores para luego subir y desaparecer. Por tanto, un valor razonable para *Min_Pts* puede ser 2692.

Basándonos en esto, para las vulnerabilidades de 2009 a 2016 y utilizando la distancia del coseno modificada, obtenemos unos valores razonables de $\varepsilon = 0{,}004$ y $Min\_Pts = 2692$.

Con este algoritmo, un ordenador con una CPU Intel i7 6700K @4.0Ghz tardó aproximadamente:

- 8 horas en computar un valor para ε que se ajuste al set de datos.
- 8 horas para calcular el valor adecuado de *Min_Pits*.
- En torno a 4 horas y 20 minutos en computar la matriz de distancias, calculando primero su mitad superior para luego espejarla, y realizar una clasificación de todas las vulnerabilidades de 2009 a 2016.

Los resultados que obtenemos con DBSCAN (tablas 6.5 y 6.6) son perfectos, pero surge la pregunta de si realmente merece la pena el tiempo que tarda el algoritmo en entrenarse, además del tiempo necesario para estimar los parámetros del algoritmo, para obtener estos resultados. Y más sabiendo que estamos empleando la mitad del set de datos. Casi se podría decir que para correr DBSCAN al completo hace falta casi un día entero sin pausa. La solución que se halló para este problema fue utilizar multi-procesamiento para la estimación de ε y *Min_Pts*. El multi-threading no es suficiente aquí, ya que el GIL no va a permitir que los threads corran simultáneamente ni aprovechar todos los cores de la CPU, por tanto es necesario pegar el "salto" de threads a procesos. Para ello creamos una especie de *main* en el módulo DBSCAN que utilizamos para estimar los parámetros. Utilizamos este segmento de código ya que lo mantenemos junto con el resto del módulo pero lo protegemos dentro del main, para que no se ejecute cuando utilizamos alguna función del módulo desde fuera.

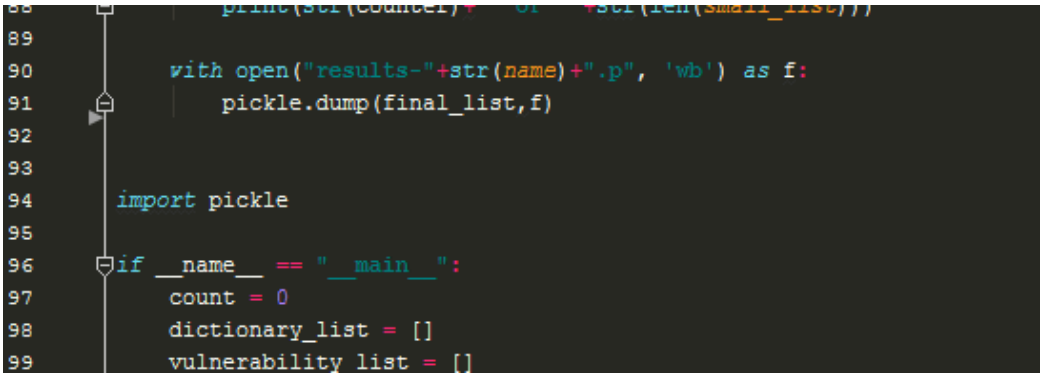


Figura 6.5: *Main* con multi-processing que estima los parámetros de DBSCAN

Lo que hacemos es ejecutar las mismas pruebas que antes (prueba de los vecinos más cercanos y los vecinos en dicha distancia) pero partiendo la lista de vectores en 4 y ejecutando 4 procesos simultáneos. Cada uno de ellos calculará las distancias y demás vecinos cercanos de su cuarto de lista respecto de la lista completa, para luego juntar los resultados y obtener la estimación global. Con esto, se consiguió reducir enormemente la duración de la estimación de cada uno de los parámetros. De 8 horas, pasamos a aproximadamente 2 horas y 45 minutos, lo que hace un total de 5 horas y media frente a las 16 de antes. Una mejora de más del 50 %.

6.3. Módulos de aprendizaje supervisado

6.3.1. Módulo K nearest neighbors

Knn es un algoritmo de Machine Learning supervisado que realiza una clasificación de los datos en función de las k observaciones más similares que tenga cada uno de ellos. Tal y como comentamos en el apartado 3.2.1, Knn necesita una base de datos ya etiquetados para poder trabajar, por tanto, no podemos utilizar Knn como herramienta de clusterización.

Con Knn validaremos los resultados de K-means y demás algoritmos de clustering de la manera que explicamos brevemente al final de la sección 6.2.1:

1. Utilizamos K-means para realizar una clasificación de todo el set de datos.

2. Entrenamos Knn con parte del set de datos, dejando lo restante como parte de un *conjunto de validación*.

3. Utilizamos Knn entrenado para clasificar el conjunto de validación, y contamos el número de veces que realiza la misma clasificación que realizó K-means en el primer paso.

El objetivo es obtener un alto porcentaje de aciertos, ya que entonces como dijimos antes, nos aseguramos de que la clasificación que hemos obtenido es buena, porque a partir de una fracción del set de datos podemos comprobar que realmente un punto es similar a los demás de su cluster.

El método que ejecuta el algoritmo es

```
run_knn(datalist, testlist, k)
```

el cual recibe como parámetros la lista con el set de entrenamiento, la lista con el set de validación y el parámetro k que determinará cuántos vecinos de los más próximos hay que mirar para determinar la clase de cada observación.

El principal problema que presenta Knn es obviamente la elección del valor k, es decir, cuántos vecinos del punto a clasificar se tienen en cuenta para elegir la clase del mismo. Generalmente, se suele elegir un valor no superior a 10, ya que entonces dependiendo del set de datos original podemos cometer errores si hay clases con pocos datos en ellas.

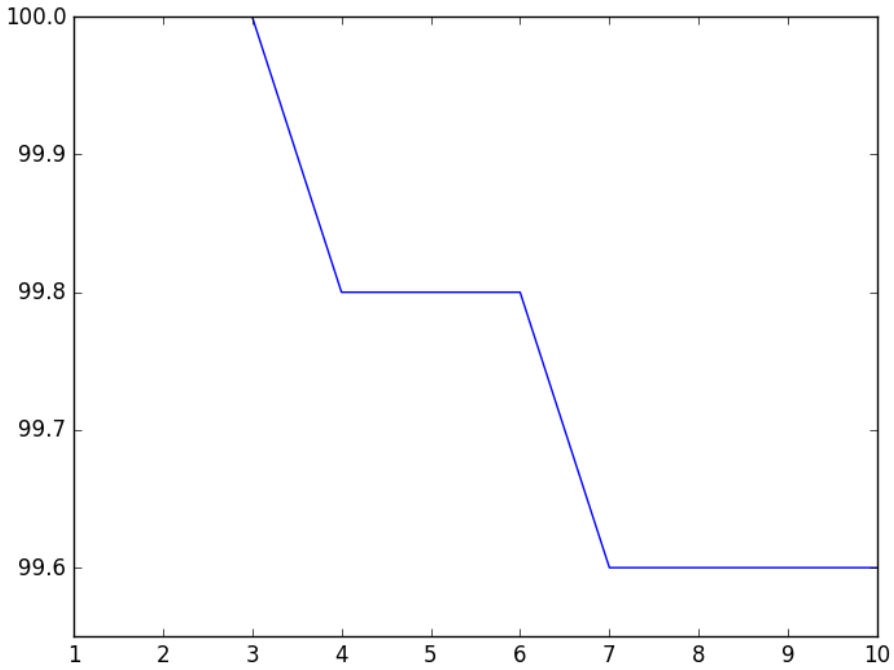


Figura 6.6: Knn, con k=1..9 clasificación vulnerabilidades de 2016

Otro problema que presenta Knn es la complejidad computacional. Es un algoritmo bastante costoso, ya que realiza los siguientes pasos:

- Para elegir los k vecinos más cercanos se calculan la totalidad de las distancias entre el punto a clasificar y todos los puntos del set de entrenamiento.
- Después hay que ordenar dicha lista y coger los k primeros elementos.
- Finalmente, se recorren esos k elementos buscando la moda entre sus valores.

En computación monohilo, esto implica un tiempo de ejecución bastante importante si el número de vulnerabilidades a clasificar es grande. Con este algoritmo, un ordenador con una CPU Intel i7 6700K @4.0Ghz tardó aproximadamente 22 minutos en realizar una clasificación de 3000 vulnerabilidades escogidas aleatoriamente habiéndo sido entrenado el algoritmo con todas las demás vulnerabilidades del set de datos.

6.3.2. Módulo SVM

Las máquinas de soporte vectorial (*Support Vector Machines*, SVMs), son un conjunto de algoritmos de Machine Learning supervisado. Dados una serie de ejemplos ya clasificados, la SVM intenta crear una separación entre dos conjuntos del espacio mediante una línea recta, un plano recto o un hiperplano n-dimensional, como se detalló en la sección 3.2.2. Dependiendo de un parámetro que se pase al método, se utilizará un kernel u otro.

Para la ejecución de este algoritmo se crea un nuevo módulo con la función

`run_svm(datalist, testlist, kernel, gamma, deg, r)`.

Todos los parámetros del método tienen un valor por defecto predefinido, y dependiendo del kernel especificado sólo se tienen en cuenta aquellos parámetros que realmente son relevantes para dicho kernel, por lo que no es necesario aportar todos los parámetros a la hora de ejecutar el algoritmo.

Este módulo no incluye una implementación propia de una máquina de soporte vectorial, si no que aprovecharemos los paquetes que nos proporciona Anaconda para ejecutar el algoritmo. Concretamente utilizaremos el paquete *sklearn* que incluye un módulo con los métodos necesarios para crear y entrenar una SVM. El único requisito necesario según la documentación de la librería es que los datos deben tener un formato específico, por lo que antes de realizar las llamadas a los métodos realizamos ese preprocesado de las dos listas de datos recibidas, la *datalist* con los datos para entrenar el modelo y la *testlist* con los datos para validar el modelo.

El módulo por defecto seleccionará un kernel lineal salvo que se pase otro parámetro que identifique qué kernel se quiere utilizar. El algoritmo soporta los siguientes:

- *linear*: Kernel lineal. En el espacio 2D, se asemejaría a intentar separar los puntos empleando una recta. No necesita parámetros.
- *rbf o Gaussiano*: Es el kernel que se emplea por defecto. Necesita un parámetro γ, que por defecto toma el valor 0.125. Ecuación 6.1
- *poly*: Kernel polinomial. Necesita el grado del polinomio, por defecto es 3, y de un término independiente, *coef0*, que por defecto es 0.
- *sigmoid*: Utiliza la función del sigmoide. Necesita un parámetro gamma, por defecto es 0.125, y un término independiente, *coef0*, que por defecto es 0.

$$f(x) = \exp\left(\frac{-\|x - x'\|^2}{2\sigma^2}\right), \tag{6.1}$$

$$\gamma = -\frac{1}{2\sigma^2}. \tag{6.2}$$

6.4. Módulos interfaz gráfica

La aplicación dispone de dos interfaces gráficas. Una para entrenar el algoritmo, clasificar las vulnerabilidades, validar los resultados y guardarlos en disco. Otra para la finalidad real de la aplicación, es decir, recibir como entrada una serie de vulnerabilidades sin clasificar y a partir de la clasificación ya existente intentar predecir a qué grupos pertenecen dichas vulnerabilidades nuevas. Empezaremos con la interfaz de entrenamiento y validación.

6.4.1. Interfaz de entrenamiento y validación

La interfaz consiste en un panel con varias pestañas, botones y un cuadro de texto:

- La pestaña de la derecha sirve para seleccionar los diccionarios que queremos que se carguen a la hora de ejecutar el algoritmo que sea. Como dijimos antes, las vulnerabilidades se agrupan el diccionarios que van por años. Dichos diccionarios tienen que estar en la misma carpeta que el proyecto, dentro de la carpeta dictionaries, siguiendo la estructura de nombre: *VulnDictionary_XXXX.p*, donde XXXX es el año del diccionario. Si alguno de los ficheros está dañado o no existe, la aplicación crea el diccionario de cero y lo actualiza con los datos más recientes.

- El panel de pestañas izquierdo es el encargado de realizar las clusterizaciones sobre el conjunto de los datos. Se elige la pestaña correspondiente al algoritmo deseado, se introducen los parámetros y se pulsa el botón de "clusterize". En el cuadro de texto inferior se irá mostrando el progreso. El funcionamiento de las pestañas de validación es similar.

- Siempre que se desee, se puede activar la casilla PCA, para que se aplique la reducción de la dimensionalidad al conjunto de los datos. Se puede especificar o bien la dimensión directamente o un *threshold* para la retención de la variabilidad, en cuyo caso se calculará una dimensión adecuada para dicho límite. El parámetro de la dimensión tiene preferencia frente al del *threshold*.

Por dentro la interfaz gráfica se compone de dos módulos, y está hecha con PyQt 4 y QtDesigner, ya que permiten utilizar un editor gráfico para la interfaz (figura 6.7) y luego es fácilmente exportable a Python. Tras ser diseñada con QtDesigner, la interfaz se guarda y se exporta a un fichero .ui, un fichero con una estructura xml que incluye todos los elementos que se utilizaron en

el editor gráfico de la interfaz. Acto seguido, se abre la consola/terminal y se utiliza el siguiente comando `pyuic4 fichero.ui -o ficheroFinal.py`, con el objetivo de traducir el contenido del archivo .ui a un script de Python. El fichero que se obtiene es la implementación de la interfaz gráfica, pero hay que hacer un fichero aparte con una clase que genere dicha interfaz, además de establecer las señales entre botones y los procedimientos que activan. Para aprovechar todos los "prints" que se realizaban en los scripts de los algoritmos, se implementó también una solución para redirigir la salida estándar al cuadro de texto de la interfaz.

El principal problema que se presentó al implementar la interfaz es que al ejecutar un algoritmo desde la misma, el hilo de ejecución se dedica a correr los métodos de los algoritmos, y no a mandar los prints por el emisor que se implementó para redirigir la salida estándar. Para solucionarlo, se creó una clase interna en uno de los módulos de la interfaz que representa un *worker*. El *worker* hereda de QObject (objeto de PyQt) y al arrancar la aplicación es movido a un QThread (sistema de multi-threading de PyQt), para a través de un sistema de señales, ejecutar el algoritmo que se le especifique con los parámetros pertinentes. Una vez terminado, el *worker* manda una señal al thread principal para indicar que ha finalizado. Si el *worker* acaba de finalizar un algoritmo de clasificación (como Knn o una SVM), el main thread ejecuta un método para representar gráficamente una matriz de confusión de los resultados.

Este sistema se empleó también porque QPixmap (la clase de Qt que gestiona los plots y las imágenes) no permite que un thread que no sea el main realice un plot. Por tanto, había que comunicar de alguna manera el thread principal y el thread con el *worker* para que se pasaran la información y mediante el uso de señales se avisara al main para que hiciera el plot de la matriz de confusión. Este problema fue el principal causante que hizo que se eliminara de la aplicación final el plot del clustering jerárquico, moviéndolo a un método aparte que se ejecuta desde el propio módulo.

Una matriz de confusión es una matriz de $n \times n$ para n clases que representa cómo se han clasificado los datos respecto de la clase a la que realmente pertenecen. Si se clasificaron correctamente o si se han cometido errores y se catalogaron como otra clase. La idea es similar a la de los falsos positivos, falsos negativos, etc. El objetivo es que la matriz quede coloreada solo en la diagonal, lo que querrá decir que todos (o casi todos) los datos se clasificaron en la clase a la que realmente pertenecen.

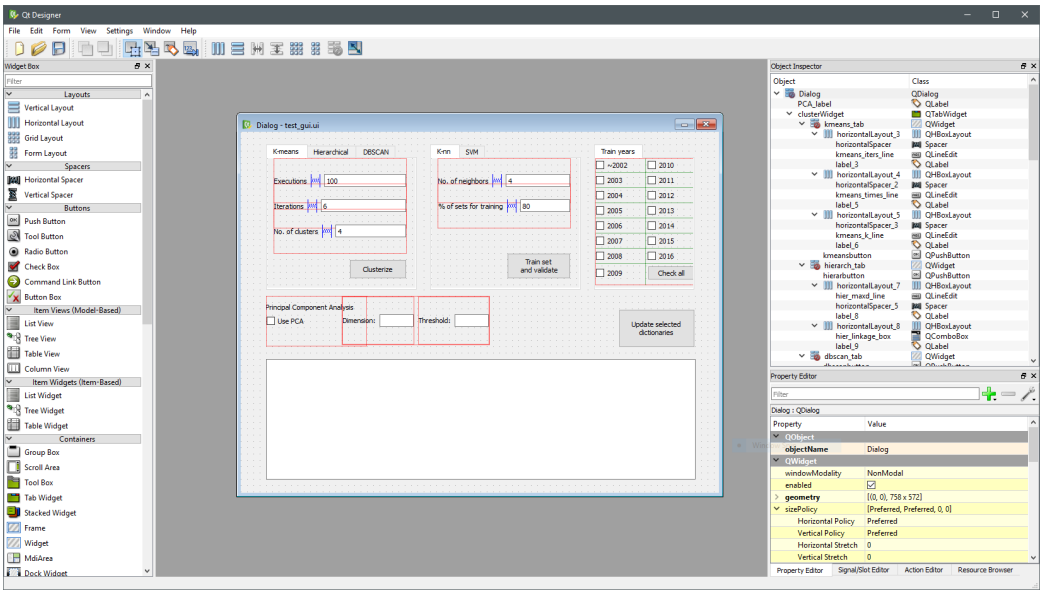


Figura 6.7: Diseño de la interfaz de entrenamiento y validación

El GIL de Python no permite que dos threads estén corriendo al mismo tiempo, pero como Numpy tiene una gran presencia en los algoritmos y suele liberar el GIL, la interfaz puede tomar el control para ir imprimiendo el texto pertinente por pantalla. Además, se incluyen los mecanismos de sincronización necesarios para asegurar las operaciones de entrada y salida, además de limitar el número de threads simultáneos a dos, para evitar que en el mismo momento haya algo más corriendo aparte de la interfaz y un algoritmo.

6.4.2. Interfaz de predicción

La interfaz de predicción es muy similar a la de entrenamiento y validación, eliminando los paneles de entrenamiento del sistema por uno para la entrada de vulnerabilidades a predecir. El cuadro de texto se ha desplazado a la derecha y está en vertical.

- La pestaña de la derecha sirve para seleccionar los diccionarios que queremos que se carguen para entrenar el algoritmo de clasificación. Su funcionamiento es igual al de la interfaz de entrenamiento y validación. Se ha eliminado el botón de actualización de los datos ya que no es un botón necesario para el usuario final.

- En el panel de pestañas central es donde están los botones y parámetros de los algoritmos de clasificación. Se ha eliminado el campo del

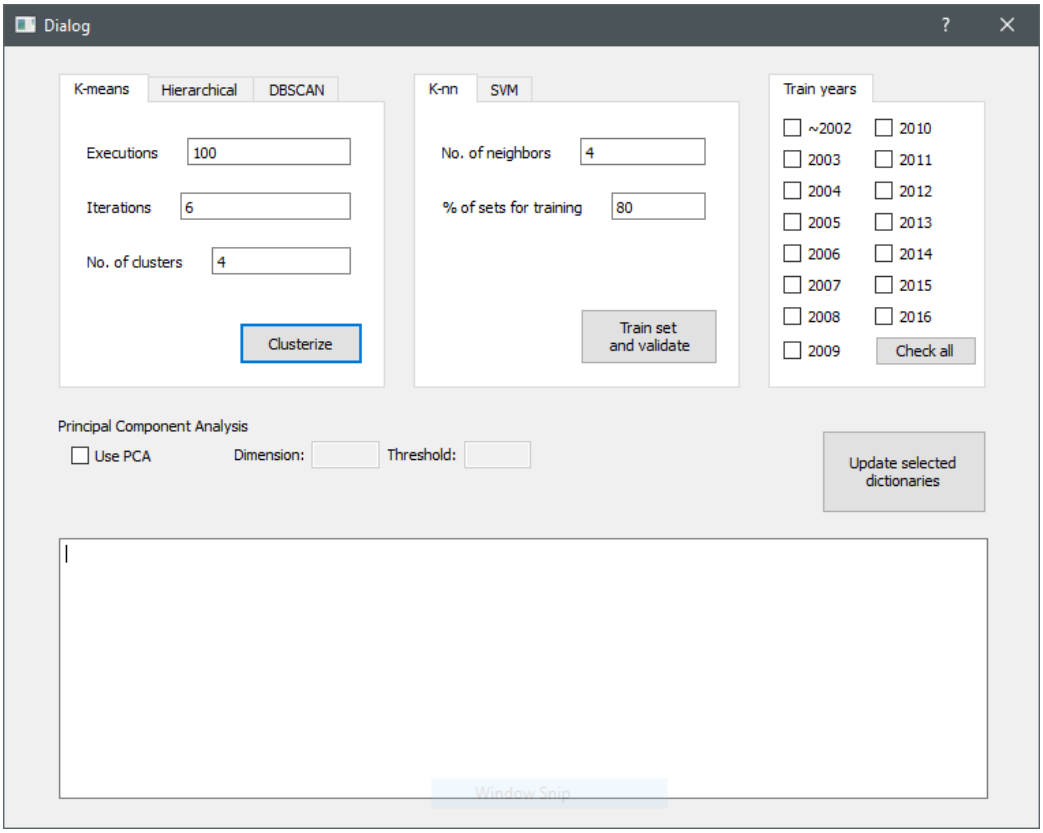


Figura 6.8: Interfaz gráfica de entrenamiento y validación corriendo con Python

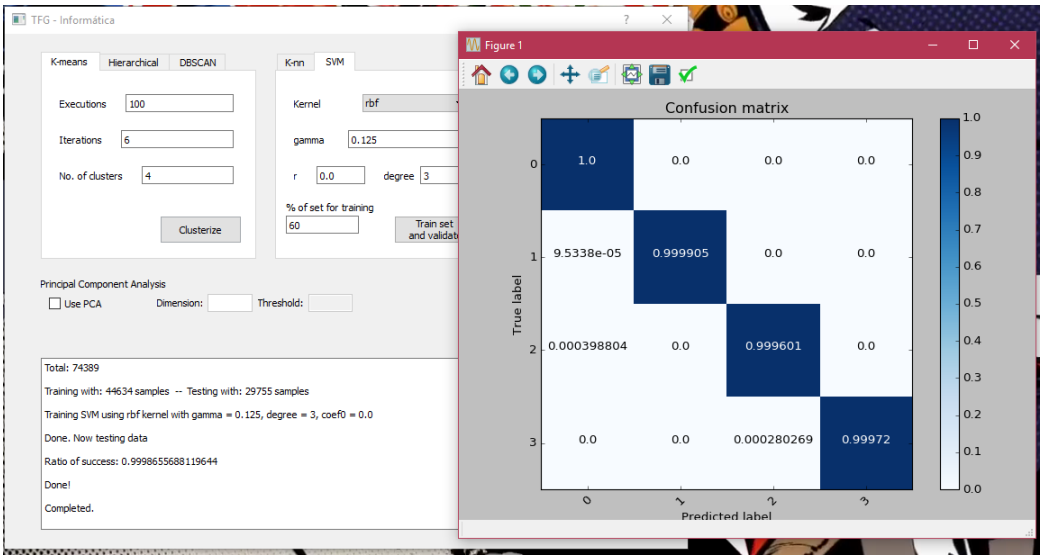


Figura 6.9: Matriz de confusión tras la validación con una SVM

porcentaje del set destinado al entrenamiento, ya que a la hora de la clasificación para vulnerabilidades futuras entrenamos el algoritmo con toda la información disponible.

- Al igual que antes, se dispone de una casilla para activar PCA.
- El panel izquierdo es el encargado de la entrada de las vulnerabilidades que se quieren clasificar. Se pueden especificar uno a uno los parámetros de la vulnerabilidad, introduciendo los valores en los campos, o bien se puede especificar el nombre de un archivo .xml que contenga las vulnerabilidades a clasificar. Es importante que el fichero .xml tenga la misma estructura que los ficheros .xml de los feeds del NIST, ya que se reutiliza código para extraer la información del fichero.
  Si se han cargado varias vulnerabilidades desde un .xml, los resultados de las clasificaciones de cada una de las vulnerabilidades se guardan en ficheros .txt, para no saturar el cuadro de texto. Si solo se clasifica una vulnerabilidad, se imprime el resultado en el cuadro de texto.

Los resultados de la clasificación muestran para cada una de las vulnerabilidades a clasificar: su ID, su vector, el grupo en el que se la ha clasificado, así como 3 vulnerabilidades con su descripción que son representativas de dicho grupo.

Por dentro, el funcionamiento es también muy similar al de la interfaz de entrenamiento y validación. Reutilizamos el mismo sistema de QThreads,

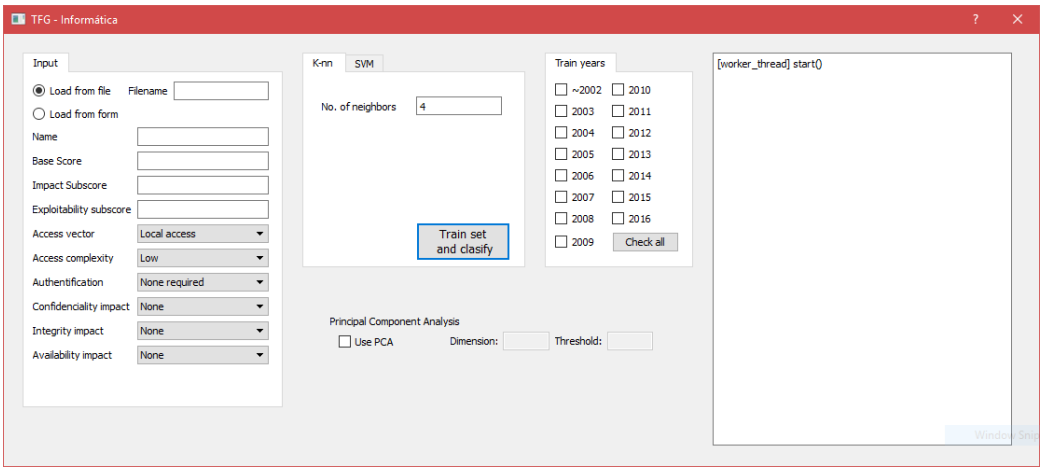


Figura 6.10: Interfaz gráfica de predicción corriendo con Python

workers y señales, solo que esta vez habrá menos algoritmos que poder ejecutar, reaprovechamos los métodos de la carga de los datos para los años seleccionados, así como los de PCA. La parte nueva de este método es la carga de los datos a clasificar, en función de dos *radio button* y de los campos que estén rellenados.

6.5. Otros módulos

6.5.1. Módulo PCA

El Análisis de Componentes Principales, ACP, (Principal Component Analysis, PCA, en inglés), es un algoritmo de Machine Learning para reducir la dimensionalidad de un set de datos. En este proyecto se empleará para reducir la dimensionalidad de los datos y comprobar si esta reducción supone un impacto importante en el tiempo de ejecución de los algoritmos o en la validación de sus resultados.

Este módulo permite ejecutar PCA sobre un determinado set de datos que recibe como lista. El módulo contiene numerosas funciones, pero la que ejecuta el algoritmo en sí es `pca(datalist, d, threshold)`. El método debe recibir el propio set de datos, y como parámetros opcionales a recibir se tienen la dimensión a la que reducir el conjunto de los datos y el valor límite para la retención de la variabilidad. En el caso de no saber a ciencia cierta a qué valor reducir la dimensionalidad, no se facilitará dicho parámetro y en su lugar se proporcionará un *threshold* o límite para calcular un valor adecuado de la dimensión reteniendo la variabilidad lo deseado. Si se especifica manualmente la dimensión a la que reducir los vectores de los datos, no será necesario incluir ningún tipo de límite o *threshold*.

Antes de realizar PCA hay que normalizar los datos centrándolos respecto de su media. Para ello, utilizamos un método que crea una nueva lista a partir del set de datos original y se va llenando con las vulnerabilidades de dicho set pero cambiando su vector por uno normalizado.

Después creamos la matriz de datos M, utilizando la información ya normalizada. Esta matriz será el parámetro que recibirá el método SVD, proporcionado por el paquete de álgebra lineal de NumPy, el cual nos generará los datos necesarios para realizar PCA: una matriz U de autovectores, otra matriz Σ con los valores singulares en la diagonal, y otra matriz V. Tal y como se explicó en el capítulo 3, para más detalles acerca de la demostración del teorema y la descomposición en valores singulares, consultar la memoria de mi Trabajo de Fin de Grado de Matemáticas [19].

Si no se aportó la dimensión a la que reducir los datos, se utiliza un método llamado `variance_retention(threshold, s)`, que utilizará el límite especificado para la retención de la variabilidad y la diagonal de la matriz de autovalores para calcular la dimensión adecuada de los datos. Una vez ob-

tenida, se seleccionan las d primeras columnas de la matriz de autovectores que generó SVD y se multiplican los vectores normalizados de los datos del set por las columnas extraídas de dicha matriz. Si se revisan las dimensiones de dicho producto de matrices, se comprobará que el resultado es el vector de los datos en la dimensión deseada.

6.5.2. Módulo distances

Este módulo es muy simple. Su función es la de hacer de librería en la que incluir todos los tipos de distancias necesarias para los algoritmos y métodos que las necesiten. Se calculan dados un par de arrays de NumPy. Entre las disponibles podemos encontrar la distancia usual (euclídea) o la distancia del coseno en $\mathbb{R}^n$ con una ligera modificación para características cualitativas. Esta modificación lo que hace es normalizar las diferencias cuando una de esas características cualitativas toma valores diferentes. Es decir, hace que las distancias entre "*Bajo - Medio*", "*Medio - Alto*z "*Bajo - Alto*"sea la misma.

6.5.3. Módulo loadAndSave

Este módulo incluye cinco métodos para la carga y guardado de diccionarios en disco, así como para guardar en disco los resultados de la clusterización de cualquiera de los tres algoritmos descritos anteriormente.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo vamos a recopilar toda la información que se obtuvo durante los capítulos anteriores para sacar conclusiones e intentar responder a todas las cuestiones que se nos han ido presentando: qué método puede ser el más adecuado para este estudio, confirmar si el clustering jerárquico no es el adecuado, si PCA realmente supone una diferencia, por ejemplo. Una vez determinado el mejor algoritmo y la mejor validación, procederemos a realizar una serie de matrices de confusión para dar un veredicto final sobre los resultados. Así mismo, se comentarán los logros conseguidos y el trabajo futuro para esta aplicación.

7.1. Conclusiones del estudio

Lo primero que se hay que hacer es juntar la información en nuevas tablas, la 7.1, la 7.2 y la 7.3.

Algoritmo	Entrenamiento (sin PCA)	Entrenamiento (con PCA)
K-means	56 minutos	54 minutos
Clustering jerárquico	157 minutos	151 minutos
DBSCAN	264 minutos	262 minutos

Tabla 7.1: Tiempos de entrenamiento para cada algoritmo

Lo primero que podemos apreciar en la tabla 7.1 es que DBSCAN es el algoritmo que más tarda en realizar la clusterización, más de cuatro veces lo que tarda el algoritmo más rápido, K-means. Además, la clusterización de DBSCAN y el clustering jerárquico se ha hecho con la mitad del set de datos

Algoritmo\Validación	Knn (k=4)	SVM (lineal)	SVM (rbf, $\gamma = 0{,}125$)
K-means (sin PCA)	27 minutos	0.822 segundos	1.33 segundos
K-means (con PCA)	25 minutos	1.54 segundos	1.67 segundos
Clustering jerárquico (sin PCA)	25 minutos	0.51 segundos	0.61 segundos
Clustering jerárquico (con PCA)	25 minutos	1.11 segundos	0.89 segundos
DBSCAN (sin PCA)	23 minutos	0.25 segundos	0.3 segundos
DBSCAN (con PCA)	22 minutos	0.92 segundos	0.88 segundos

Tabla 7.2: Tiempos de validación para cada algoritmo

Algoritmo\Validación	Knn (k=4)	SVM (lineal)	SVM (rbf, $\gamma = 0{,}125$)
K-means (sin PCA)	99.967 %	99.995 %	99.91 %
K-means (con PCA)	99.885 %	99.992 %	99.993 %
Clustering jerárquico (sin PCA)	99.86 %	99.846 %	99.969 %
Clustering jerárquico (con PCA)	99.783 %	99.97 %	99.967 %
DBSCAN (sin PCA)	100 %	100 %	100 %
DBSCAN (con PCA)	100 %	100 %	100 %

Tabla 7.3: Porcentaje de acierto en la validación para cada algoritmo

por imposibilidad física del PC, por lo que se ratifica aún más la ventaja de K-means frente a los demás en lo que a tiempo se refiere.

Sin embargo, a primera vista puede surgir la pregunta de si realmente es correcto elegir K-means guiándonos solo por el tiempo, ya que es posible que sus resultados sean peores que los de los demás al tardar bastante menos. Sin embargo, si nos dirigimos a la tabla 7.3, vemos que los resultados de K-means no son tan malos, siempre y cuando no se utilice PCA. Si se junta la información de todas las tablas, podemos apreciar que en la mayoría de los casos PCA apenas aporta una reducción en los tiempos de ejecución y de hecho los resultados empeoran cuando se utiliza. Por tanto, para cualquier tipo de prueba con este set de datos es mejor no utilizar PCA para reducir la dimensionalidad.

Así mismo, en la misma tabla 7.3, se aprecia cómo efectivamente, los resultados del clustering jerárquico son los peores de los tres algoritmos, se utilice el algoritmo que se utilice para la validación. Por tanto, podemos confirmar que a pesar de que nuestro set de datos no es especialmente grande, el clustering jerárquico no es el más adecuado para grandes cantidades de datos.

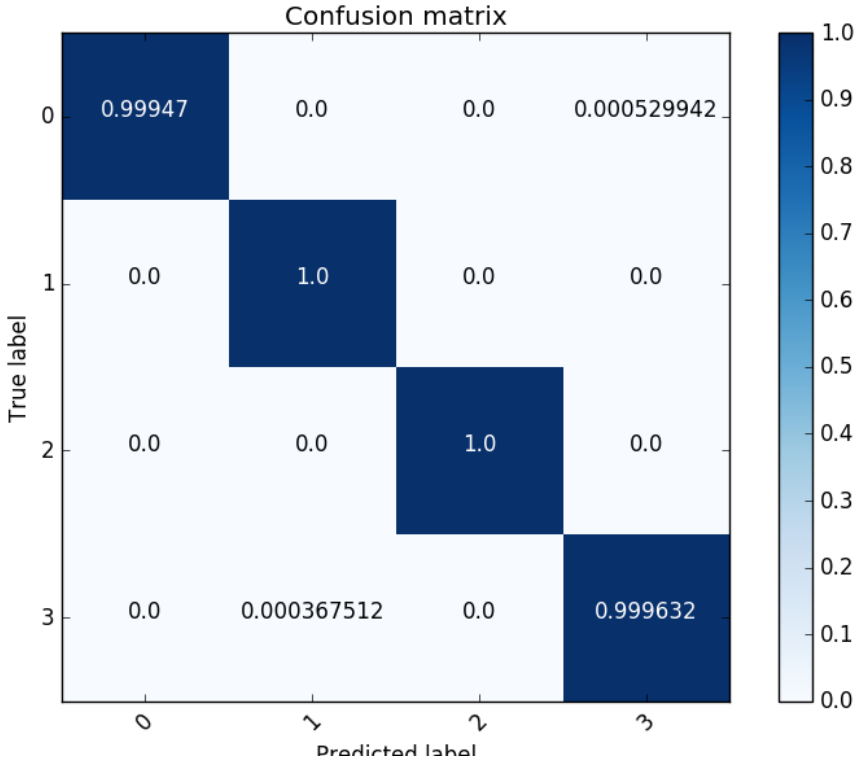


Figura 7.1: Matriz de K-means con kernel gaussiano

Luego si queremos elegir un algoritmo para nuestro set de datos, tendremos que elegir entre K-means y DBSCAN. Para ello, vamos a revisar las matrices de confusión de los distintos algoritmos, para ver los resultados desde otro punto de vista e intentar sacar conclusiones.

Para crear la matriz de confusión antes hay que elegir un método de validación. En lo que a validación se refiere, está claro que el algoritmo que mejores resultados da es la SVM, tanto con kernel gaussiano como lineal, ya que ofrecen un resultado muy similar a Knn pero tardando muchísimo menos.

Las matrices obtenidas se corresponden con las de las figuras 7.1, 7.2, 7.3 y 7.4.

Como se puede apreciar en las matrices, DBSCAN ofrece unos resultados mejores que los de K-means, aunque también tarda mucho más. En estos casos se debe entrar a valorar si de verdad merece la pena el tiempo de entrenamiento en relación con los resultados obtenidos, teniendo en cuenta

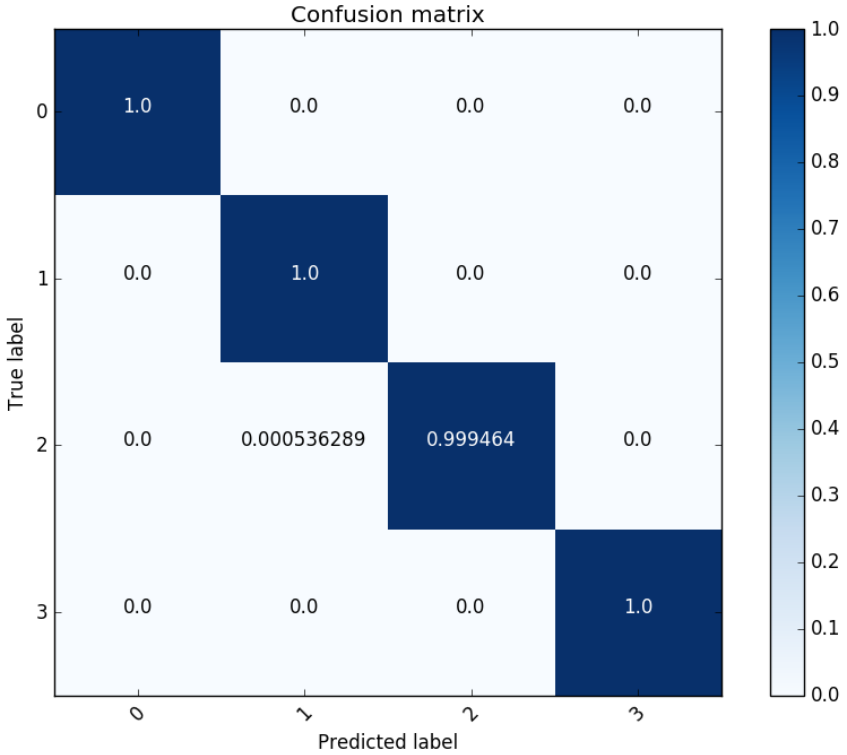


Figura 7.2: Matriz de K-means con kernel lineal

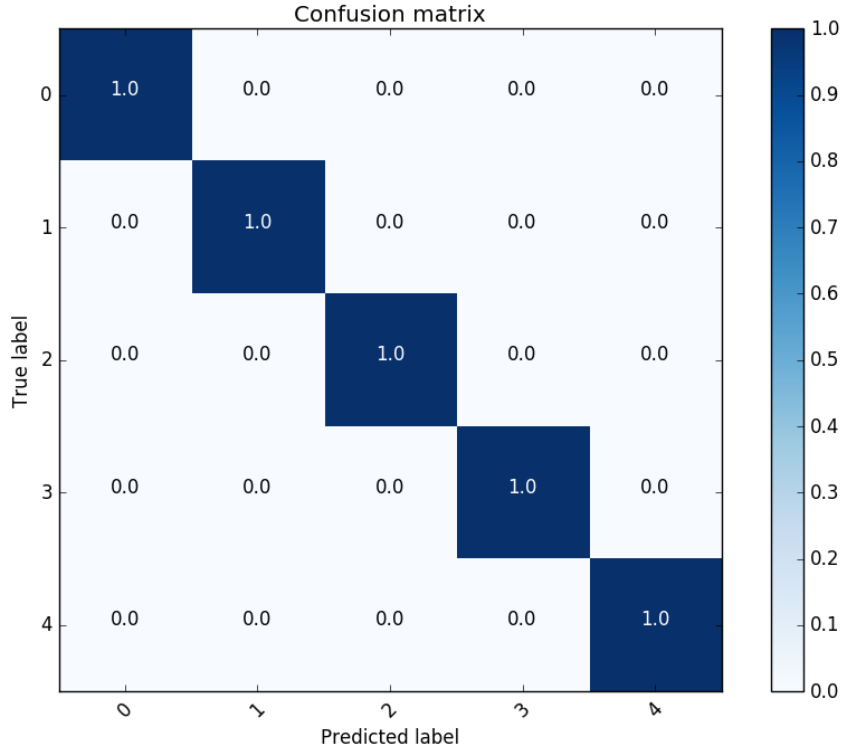


Figura 7.3: Matriz de DBSCAN con kernel gaussiano

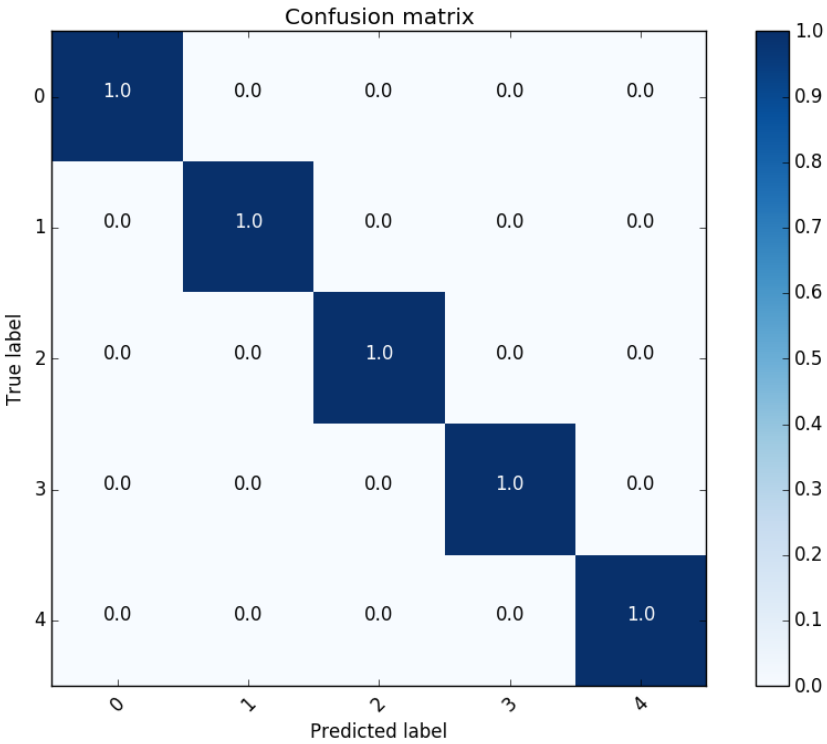


Figura 7.4: Matriz de DBSCAN con kernel lineal

cada cuánto se va a actualizar el entrenamiento del algoritmo, qué es lo que buscamos, el ritmo de crecimiento del set de datos, etc. Dado que las vulnerabilidades de la NVD no cambian tan repentinamente y que el ritmo de crecimiento es estable y no muy alto, quizás DBSCAN podría ser la solución para este estudio, ya que el sistema no va a ser entrenado a diario y es probable que con un entrenamiento mensual sea suficiente, aunque la periodicidad del entrenamiento es otro tema de estudio.

Por tanto, una vez determinados los mejores algoritmos para nuestro estudio, enumeramos las conclusiones que hemos sacado del mismo:

- K-means es uno de los algoritmos más conocidos y utilizados dado su corto tiempo de entrenamiento y sus buenos resultados, y las pruebas de este estudio lo demuestran.
- Confirmamos que el clustering jerárquico no es el más adecuado para sets de datos muy grandes.
- DBSCAN es un algoritmo que ofrece buenos resultados pero posee un tiempo de ejecución algo grande. Podría mejorarse su implementación si eliminamos la necesidad de una matriz de distancias. Para estudios como este que no requieren de entrenamientos muy frecuentes puede ser una opción razonable.
- PCA puede servir para comprimir la información y facilitar ciertos cálculos, pero una reducción de la dimensionalidad de 9 a 5 no ha ayudado mucho en este estudio.

7.2. Logros conseguidos

El principal logro obtenido en este proyecto ha sido desarrollar una aplicación íntegramente en Python que permite extraer la información de la NVD, organizarla y realizar una clusterización de los datos para predecir la clase y características de vulnerabilidades futuras. La aplicación permite utilizar varios algoritmos de clusterización y clasificación supervisada, variar los parámetros de dichos algoritmos, probar diferentes combinaciones y permite diferentes entradas para la predicción de características de vulnerabilidades.

Además, se han adquirido conocimientos y conceptos de Machine Learning no supervisado, algoritmos de clusterización, técnicas de clasificación, así como multi-threading y multi-processing empleando señales en Python y la programación de interfaces gráficas utilizando Qt.

7.3. Trabajo futuro

Llegado este punto del trabajo, se pueden plantear una serie de posibles mejoras futuras que permitan extender los logros alcanzados y mejorar la aplicación:

- Mejorar la implementación de DBSCAN. La principal desventaja que presenta DBSCAN es su implementación en Anaconda y la necesidad de una matriz de distancias. Una posible mejora podría ser reimplementar cierta parte del algoritmo para que no necesite de dicha matriz para calcular las vecindades de los datos, sino que utilizara una base de datos o similar.

- Control más específico de las fechas. Las vulnerabilidades de la NVD vienen clasificadas en diccionarios por años, y aunque es un nivel de organización suficiente para realizar este estudio, dar más control al usuario para que pueda escoger un rango de fechas más específico, especificando días o meses, sería bueno.

- Incluir una mayor variedad de algoritmos entre los que elegir. Para este estudio se ha intentado realizar una selección de los algoritmos de clusterización clásicos más conocidos y utilizados, sin embargo, la inclusión de otros algoritmos con enfoques más modernos como aquellos que emplean grafos y comunidades podría ser interesante.

- Incluir un botón en la interfaz para reiniciar los datos. La interfaz no dispone de un botón para dejar los datos "limpios", tal y como se descargaron de la NVD. Sería bueno incluir un botón más en la interfaz que permitiera hacerlo.

- Conseguir la portabilidad a Mac. En el momento de escribir esta memoria la aplicación no funciona en Mac. Dispongo de un portátil MacBook Air y el mismo código con los mismos datos de la NVD se ejecuta sin problemas en sistemas Windows o Linux pero no en Mac. Cuando NumPy intentar calcular la norma de un vector se producen errores y devuelve números muy grandes o *NaN*. No se ha podido identificar la raíz de estos errores, aunque preveo que puede tener relación con la precisión de los floats y cómo los gestiona el procesador del Mac. Si se consiguiera arreglar este problema para conseguir una total portabilidad de la aplicación sería perfecto.

- Una mejor depuración en la aplicación. Muchos de los cuadros de texto de las interfaces gráficas controlan que el contenido de los mismos sea

del tipo de datos adecuado (*float, int, string...*), pero no todos controlan que sean valores razonables. Es decir, podría especificarse para una vulnerabilidad un score de -8.465, por ejemplo, cosa que es imposible. Por tanto, una mejora posible sería controlar más exhaustivamente los valores que pueda introducir el usuario.

- Utilizar CUDA o alguna otra herramienta de paralelización para agilizar la ejecución de los algoritmos. Al realizar los tests del método del codo se empezó a ver el principal inconveniente del Machine Learning, el tiempo de entrenamiento de los algoritmos. Las licencias de pago de Anaconda incluyen un paquete que permite acelerar código en Python sin apenas modificarlo, que era exactamente lo que se buscaba para este proyecto, ya que no tengo ningún conocimiento sobre la sintaxis y el funcionamiento de CUDA, pero el alto coste económico de dichas licencias hizo que se desechara totalmente la idea. Finalmente se terminó por utilizar algo de programción multihilo y multiproceso para acelerar la estimación de los parámetros de cierto algoritmo, pero sería bueno incluir algún tipo de herramienta de paralelización para acelerar la ejecución de los algoritmos en sí.

Capítulo 8

Manual de instalación y uso

En este manual se va a explicar el funcionamiento de la aplicación desarrollada. La primera parte se dedicará a explicar los requisitos del sistema necesarios para la aplicación. La segunda para el proceso de instalación, para que la aplicación funcione correctamente. En la tercera parte se explicará el funcionamiento de las dos interfaces gráficas de la aplicación utilizando un caso de uso como hilo de la explicación.

8.1. Requisitos del sistema

Los requisitos del sistema son:

- Sistema operativo: Windows o Linux.
- Python 3.5.2 instalado.
- Anaconda 4.1.1 instalado.
- Conexión a internet disponible si se desea actualizar los paquetes de los diccionarios.

8.2. Instalación

Los pasos a seguir para instalar la aplicación son:

- En caso de no tener instalada la versión 3.5.2 de Python3 en el sistema, accedemos a la web de Python [30] y descargamos la última versión de Python3 para nuestro sistema.

- En caso de no tener instalado Anaconda en el sistema, accedemos a la web de Anaconda [1] y descargamos la versión de Anaconda 4.1.1. Si no aparece como la versión más reciente, accedemos al *Anaconda installer archive* (archivo de versiones, parte inferior de la página de descargas) y la descargaremos desde ahí.

- Finalmente copiamos al sistema la carpeta del proyecto. Bien desde el medio físico provisto con el Trabajo Fin de Grado o bien accediendo a mi GitHub [33] y clonando el proyecto o descargándolo en un .zip y descomprimiéndolo.

Ya se puede ejecutar la aplicación y comenzar a utilizarla. Para las instrucciones de las dos aplicaciones con interfaz gráfica, vamos a realizar el siguiente caso de uso:

- Se entrenará el sistema con las vulnerabilidades de la base de datos que sean anteriores a 2016, para luego realizar una clasificación con la intención de "predecir" detalles de las vulnerabilidades de 2016 con la aplicación de predicción.

8.3. Guía de uso de la aplicación de entrenamiento

Para abrir esta aplicación se corre el script *train_launcher.py*. La pantalla que aparezca será similar a la figura 8.1.

Lo primero de todo será actualizar la información de los diccionarios, para contar con los datos más actuales. Para ello seleccionamos todos los años pulsando el botón *Check all* y después el botón *Update selected dictionaries*. La aplicación intentará cargar los diccionarios desde la carpeta *dictionaries* del proyecto. Si encuentra un fichero dañado o falta alguno de los diccionarios de algún año seleccionado, creará el fichero desde cero. Una vez cargados todos, se llevará a cabo el proceso de actualización. Puede tardar un poco dependiendo de la velocidad de la conexión a internet y de la potencia del procesador de la máquina.

Una vez terminado el proceso de actualización de los datos se presentará un mensaje como el de la figura 8.2 , por lo que procederemos a realizar una clasificación de los datos y a validarla. Para ello mantenemos seleccionados los diccionarios que 2002 a 2015, y seleccionamos uno de los tres métodos

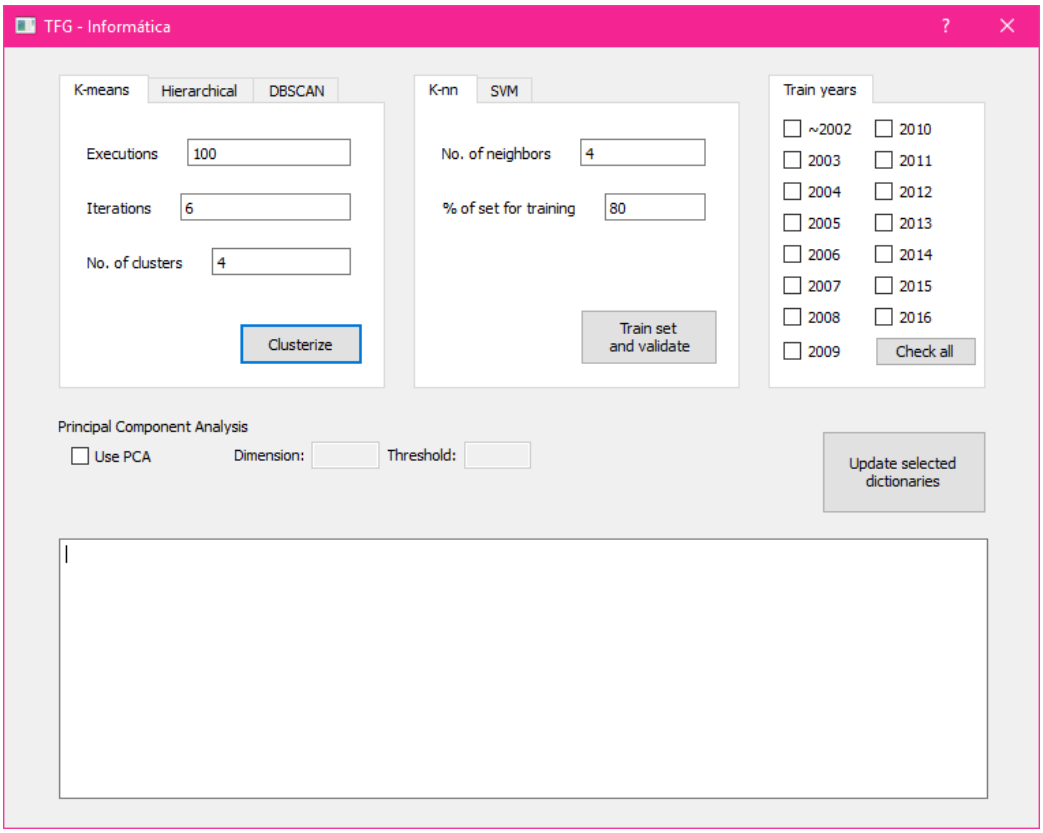


Figura 8.1: Interfaz gráfica de entrenamiento y validación

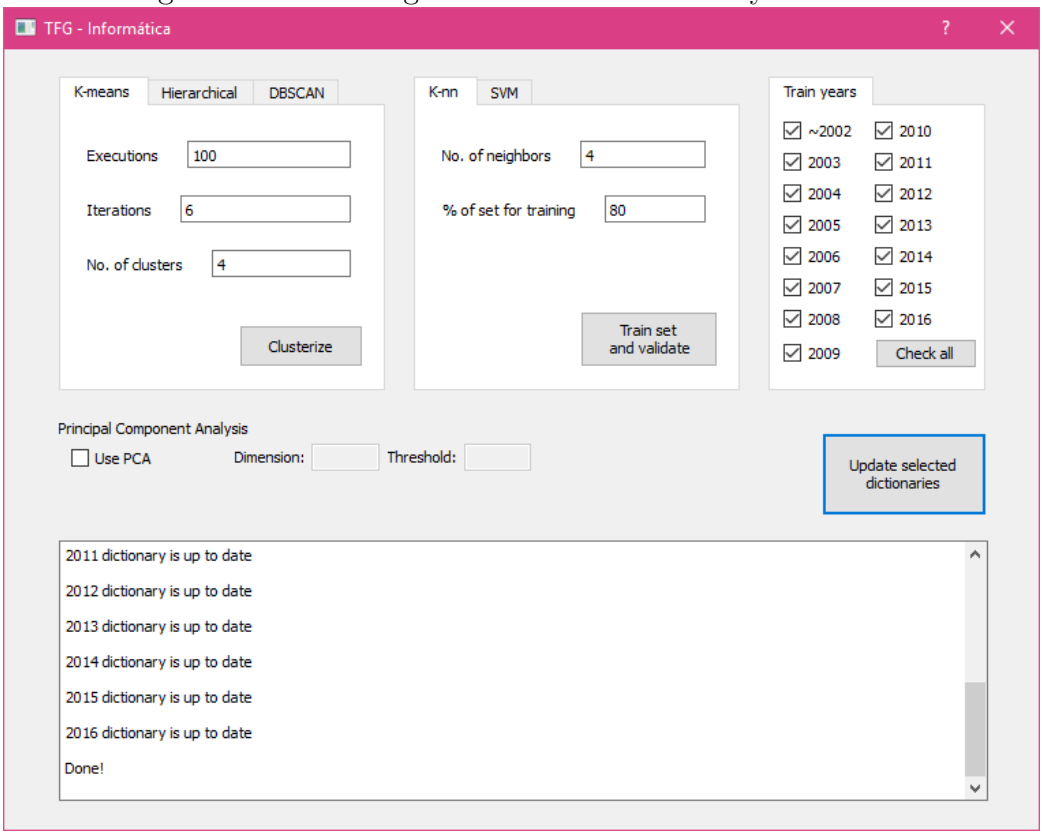


Figura 8.2: Actualización de los datos utilizando la interfaz gráfica

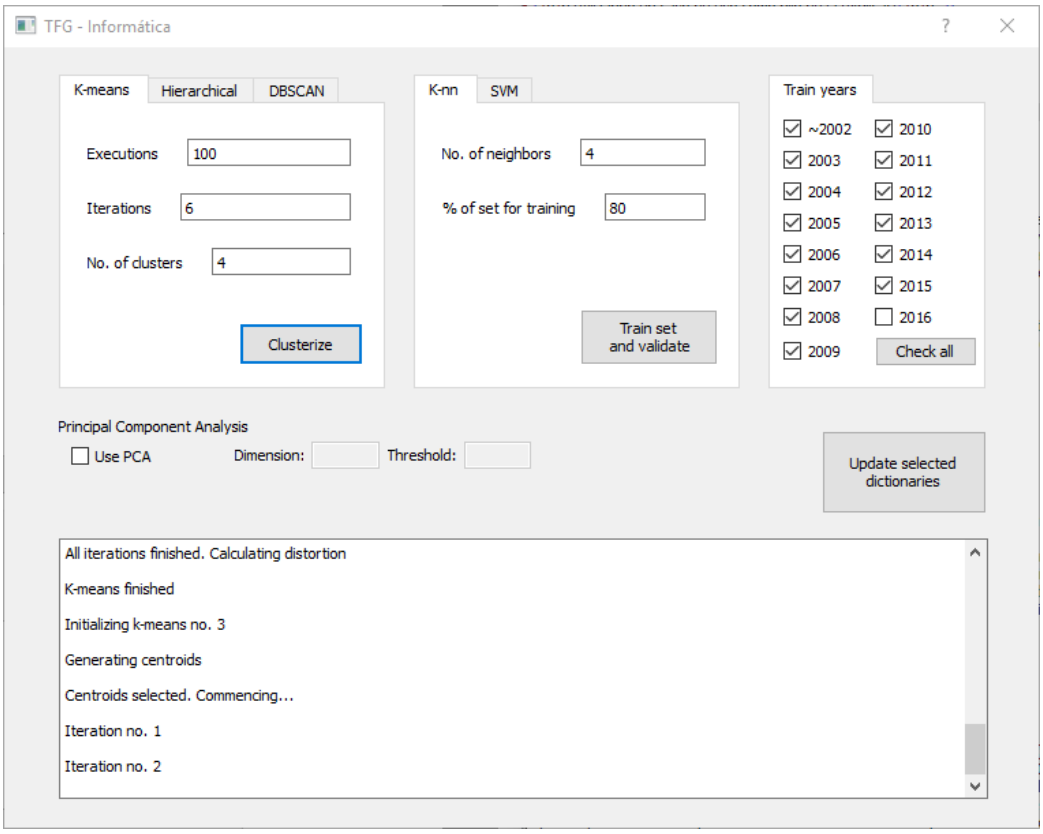


Figura 8.3: Ejecutando un algoritmo de clusterización desde la interfaz gráfica

de clusterización de la pestaña de la izquierda así como sus parámetros. Por ejemplo, K-means. Si se desea utilizar la reducción de la dimensionalidad de PCA, se puede activar la casilla de PCA e introducir la dimensión a la que reducir directamente o un límite para calcular dicha dimensión en función de la retención de ese porcentaje de la la variabilidad. En este caso, vistos los resultados del capítulo 7 y que no aporta apenas beneficios, no vamos a utilizarlo. Cuando tengamos todo seleccionado y rellenado, pulsamos el botón de *Clusterize* y esperamos a que termine el entrenamiento. Dependiendo del algoritmo elegido puede que la aplicación tarde más o menos en clusterizar los datos. Puede utilizarse la tabla del capítulo 7 como una referencia.

Una vez terminado el entrenamiento se procede a validar los resultados. Para ello mantenemos activados los mismos años que se utilizaron en el entrenamiento y nos dirigimos a la pestaña central, la de validación. Allí al igual que antes, seleccionaremos uno de los dos algoritmos para validar los resultados anteriores. Esta vez optaremos por una SVM. Podemos cambiar

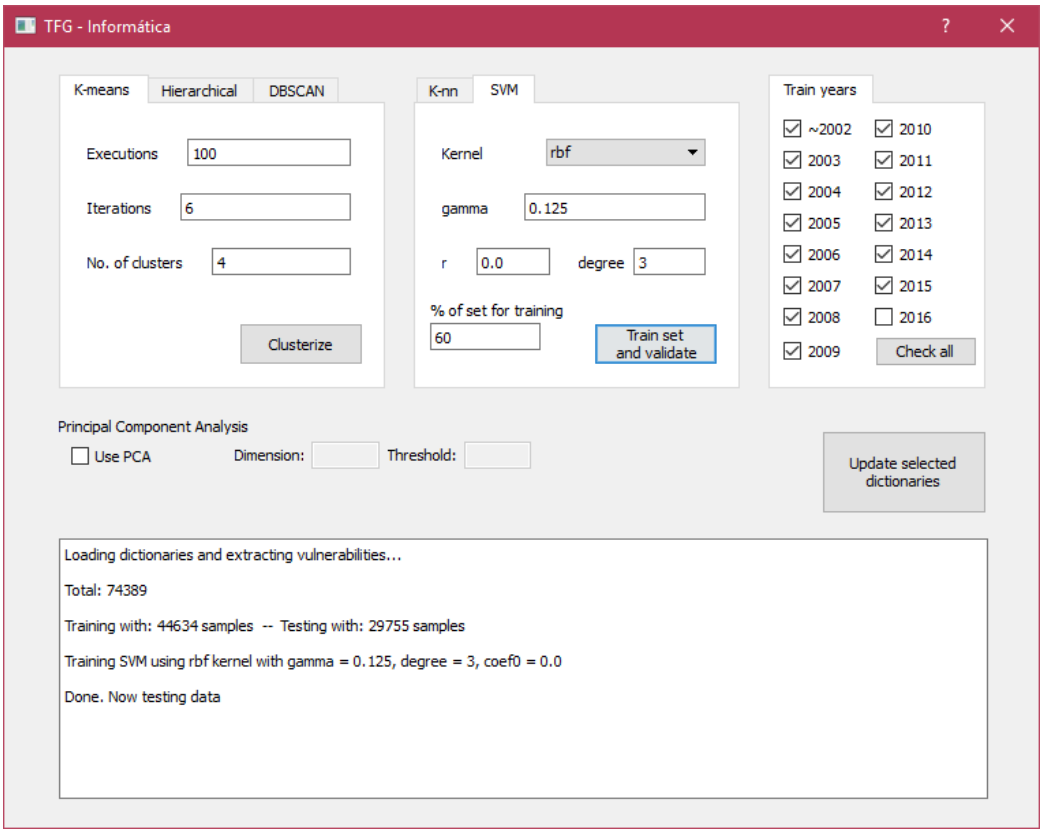


Figura 8.4: Validando los resultados desde la interfaz gráfica

el *kernel* a utilizar seleccionando uno de entre los disponibles. Para saber qué parámetros requiere cada *kernel*, puede consultarse el apartado 6.3.2. Seleccionaremos el kernel gaussiano con $gamma = 0{,}125$. Una vez introducidos los parámetros, elegimos qué porcentaje del set de datos queremos que se destine a entrenar el algoritmo supervisado. Por defecto es un 80 % pero puede variarse, aunque no se recomienda utilizar un porcentaje muy alto, ya que no se probarán apenas datos y los resultados pueden no ser reales. Especificaremos un 60 %, que es un valor razonable. El porcentaje restante del set de datos será el que se pruebe para comprobar los resultados de la clasificación previa. Así mismo, puede activarse la casilla de PCA para utilizarlo de nuevo, pero no lo haremos. Cuando todos los valores estén introducidos, pulsamos el botón de *Train set and validate* y esperamos a obtener los resultados como en la figura 8.4.

Cuando el algoritmo termine, imprimirá por el cuadro de texto el ratio general de aciertos y realizará el plot de una matriz de confusión, para mos-

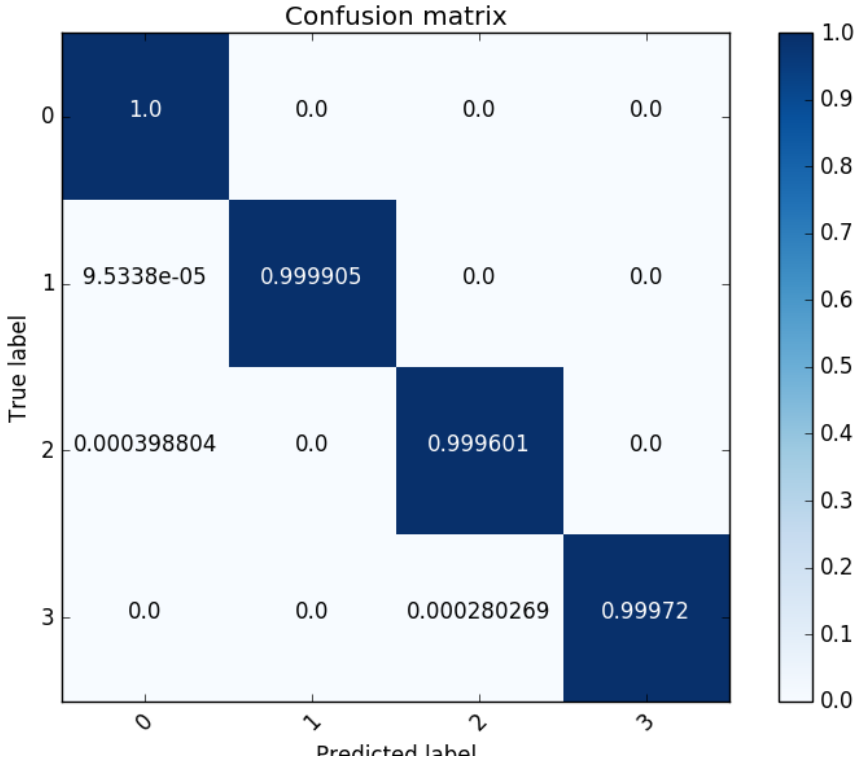


Figura 8.5: Matriz de confusión tras la validación

trar los resultados de una manera más gráfica y detallada. El objetivo de esta validación es que el algoritmo clasifique ahora los datos en la misma clase en la que fueron clasificados antes. Esto se verá representado en la matriz de confusión con la diagonal siendo la única parte coloreada.

Si los resultados no son adecuados o no son los deseados, puede realizarse nuevamente un entrenamiento con otro algoritmo o variar los parámetros. En el caso de que se desee "reiniciar" la base de datos y dejarla como estaba al principio, bastará con eliminar los ficheros de los diccionarios de la carpeta *dictionaries* y actualizarlos desde la aplicación, como se indicó al principio de esta sección.

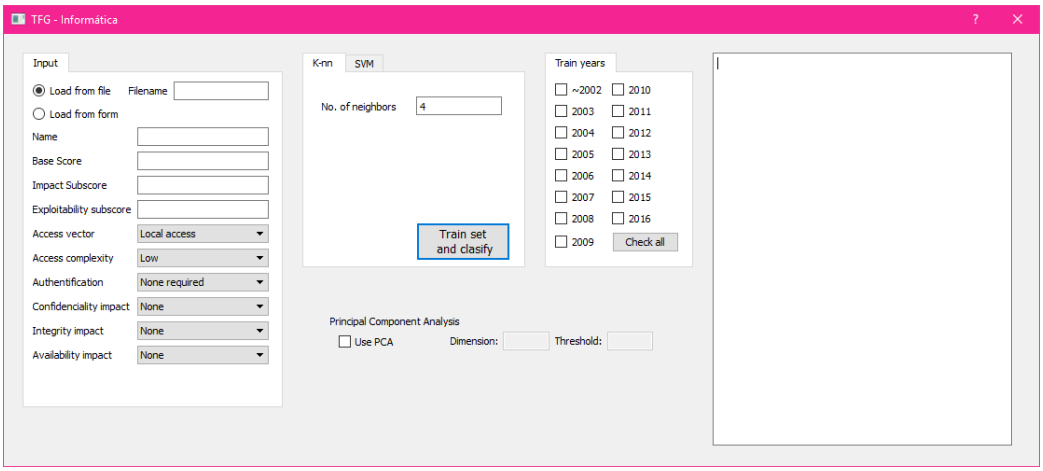


Figura 8.6: Interfaz gráfica de predicción

8.4. Guía de uso de la aplicación de predicción

Para abrir esta aplicación hay que correr el script *predict_launcher.py*. La pantalla que aparezca será similar a la figura 8.6.

Esta vez no será necesario actualizar los datos, ya que sabemos que están actualizados. Por tanto, simplemente marcaremos los años que queremos utilizar como base para la clasificación supervisada. Nos dirigimos a la pestaña central, la de validación. Allí al igual que antes, seleccionaremos uno de los dos algoritmos para intentar predecir a qué clase pertenecen las nuevas vulnerabilidades. Una vez introducidos los parámetros, nos dirijimos al panel izquierdo en el que especificaremos la entrada de la aplicación. Ahí seleccionaremos una de las dos opciones disponibles: *Load from file*, cargar desde un archivo; o *Load from form*, cargar desde formulario. En caso de seleccionar la primera, deberemos aportar el nombre del fichero, extensión incluida, .xml que contiene la información de las vulnerabilidades a especificar. Es importante que el fichero .xml tenga el mismo formato que los ficheros .xml que proporciona el NIST, ya que se tratará como uno igual. Si se selecciona la segunda opción, debe rellenarse el formulario con los datos de la vulnerabilidad a clasificar, dando valores a cada uno de los campos, incluyendo un nombre provisional. En nuestro caso, vamos a rellenar el formulario con los datos de una vulnerabilidad de 2016.

Una vez seleccionada la entrada, activamos la casilla de PCA si deseamos

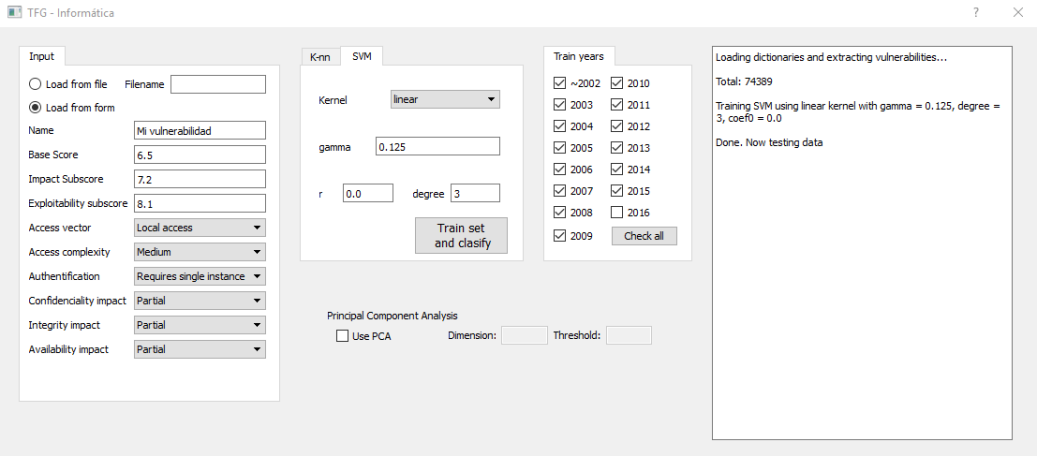


Figura 8.7: Prediciendo una vulnerabilidad usando la interfaz gráfica

utilizarlo de nuevo, y presionamos el botón de *Train set and clasify* y esperamos a obtener los resultados. Si solo queremos clasificar una vulnerabilidad, como es nuestro caso, se mostrarán los resultados de la misma en el cuadro de texto. Si son más, se crearán ficheros .txt, uno por cada vulnerabilidad, en la carpeta *vulnerabilities* del proyecto, con los resultados de cada una de ellas.

Los resultados para cada vulnerabilidad incluyen:

- Un resumen de los detalles de la vulnerabilidad a clasificar.
- El vector de la vulnerabilidad a clasificar.
- El grupo en que ha sido clasificada la vulnerabilidad.
- Tres vulnerabilidades de dicho grupo que sean representativas del mismo.

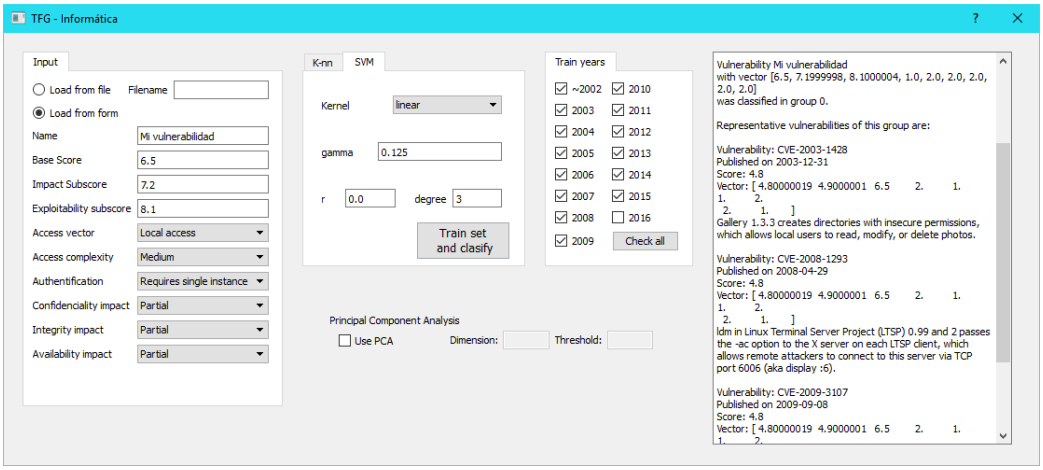


Figura 8.8: Resultados de la predicción usando la interfaz gráfica

Bibliografía

[1] Anaconda, Open Data Science Platform powered by Python
`www.continuum.io`

[2] Apache Software Fundation
`www.apache.org/`

[3] Common Vulnerabilities and Exposure, CVE
`cve.mitre.org`

[4] Common Vulnerability Scoring System
`www.first.org/cvss`

[5] Current CVSS Score Distribution For All Vulnerabilities
`www.cvedetails.com`

[6] Delta de Kronecker
`es.wikipedia.org/wiki/Delta_de_Kronecker`

[7] S. Dua y X. Du, *"Data Mining and Machine Learning in Cybersecurity"*, Editorial CRC Press, 2011

[8] El gran negocio de la ciberseguridad
`economia.elpais.com/economia/2016/06/06/actualidad/1465236317_910951.html`

[9] Emergency Patch for Windows vulnerability MS15-078
`ghacks.net/2015/07/21/emergency-patch-for-windows-vulnerability-ms15-078-rel`

[10] P. Flach, *"Machine Learning, The Art and Science of Algorithms that Make Sense of Data"*, Editorial Cambridge, 2012

[11] GIL, The Global Interpreter Lock
`wiki.python.org/moin/GlobalInterpreterLock`

[12] GitHub
`github.com`

[13] T. Hastie, R. Tibshirani y J. Friedman, *"The Elements of Statistical Learning"*, Editorial Springer, 2ª edición, 2009

[14] Heartbleed vulnerability
`heartbleed.com/`

[15] Implementación clustering jerárquico en *SciPy*
`docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage`

[16] Implementación DBSCAN en *sklearn*
`scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html`

[17] Implementación SVD en *NumPy*
`docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.svd.html`

[18] Implementación SVM en *sklearn*
`scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`

[19] Sergio Michiels Mangas, *"Fundamentos matemáticos de clasificación y predicción de vulnerabilidades en ciberseguridad basada en técnicas de Machine Learning"*, Noviembre, 2016

[20] Microsoft February 2013 Advance Notification Multiple Vulnerabilities
`www.securityfocus.com/bid/57846/discuss`

[21] Microsoft Security Bulletin MS15-021 - Critical
`technet.microsoft.com/en-us/library/security/ms15-021.aspx`

[22] Microsoft : Vulnerability Statistics
`www.cvedetails.com/vendor/26/Microsoft.html`

[23] Microsoft Windows win32k.sys Kernel Driver Privilege Escalation Vulnerability
`tools.cisco.com/security/center/viewAlert.x?alertId=22826`

[24] MITRE Corporation
`www.mitre.org/`

[25] National Institute of Standards and Technology
`csrc.nist.gov/`

[26] National Vulnerability Database
`nvd.nist.gov/`

[27] NIST - CVSS Information and about incomplete data
`nvd.nist.gov/cvss.cfm`

[28] PyCharm, Python IDE for professional developers
`www.jetbrains.com/pycharm`

[29] PyQt
`riverbankcomputing.com/software/pyqt/intro`

[30] Python
`www.python.org`

[31] Qt
`www.qt.io`

[32] QtDesigner Manual for Qt 4.8
`doc.qt.io/qt-4.8/designer-manual.html`

[33] Repositorio de GitHub con el Trabajo de Fin de Grado
`github.com/carchenilla/TFG-informatica`

[34] Vulnerabilities By Type
`www.cvedetails.com/vulnerabilities-by-types.php`