

Práctica 2: Representación de los datos
Sistemas de Información —Ing. de la Ciberseguridad
2022-2023

Pablo Pastor López, Pablo Redondo Castro, Gabriel Medrano Sanchez

16 de mayo de 2023



Índice

1. Introducción	3
2. Entorno de la práctica	4
2.1. Docker y servicios	4
2.1.1. Estructura del proyecto	5
3. Ejercicio 1	6
4. Ejercicio2	8
5. Ejercicio 3	10
6. Ejercicio 4	11
6.1. Login Page	11
6.2. Generación PDF y análisis de otras métricas	11
6.3. Analisis de otras métricas	12
6.4. Mostrar datos de algun servicio web mediante otra API	13
7. Ejercicio 5	15
7.1. Realizar un método de Regresion Lineal	15
7.2. Realizar un método de Decision Tree	17
7.3. Realizar un método de Random forest	19

1. Introducción

En esta segunda práctica de la asignatura, continuaremos trabajando en el desarrollo del sistema MIS que iniciamos en la primera práctica. Sin embargo, esta vez nos enfocaremos en la creación de un dashboard o Cuadro de Mando Integral (CMI) utilizando el lenguaje de programación Python. Este dashboard será una representación visual de los datos obtenidos tras el tratamiento realizado en la primera práctica.

La creación de este dashboard es importante porque los informes generados en la primera práctica son estáticos y no permiten personalizar los diagramas. Además, si se envían datos en tiempo real, no se pueden representar. Por ello, nuestros clientes desean que diseñemos el almacén de datos y, posteriormente, diseñemos un CMI que facilite la toma de decisiones a la dirección de la empresa.

Durante esta práctica, trabajaremos en grupos y seguiremos utilizando el mismo entorno que en la práctica anterior. Con la creación de este dashboard, podremos presentar los datos de una manera más visual y personalizada, lo que permitirá una toma de decisiones más efectiva y eficiente.

Ahora, nuestros clientes nos han pedido diseñar un almacén de datos, junto a un CMI para facilitar la toma de decisiones de la empresa:

2. Entorno de la práctica

Para el desarrollo de la práctica, hemos modificado un repositorio público de GitHub <https://github.com/app-generator/flask-soft-ui-dashboard>, el cual nos ahorra una cantidad considerable de tiempo al tener las imágenes y configuraciones de docker incorporadas.

Nuestro repositorio de trabajo sigue siendo el mismo de la práctica anterior:

https://github.com/medranoGG/SI_Proyectos/

2.1. Docker y servicios

Este proyecto contaba con dos servicios: *app-seed*, el cual hemos cambiado por *CMI*, y el servicio de *nginx*.

- El contenedor **CMI** es el encargado de gestionar el backend del proyecto
- El contenedor **nginx** es el encargado de gestionar toda la parte del frontend del proyecto

El fichero de configuración de docker del proyecto se puede encontrar como **docker-compose.yml**

Para iniciar el proyecto, podemos utilizar el siguiente comando: *docker-compose up -build*

El servicio se levanta en *localhost:5085*

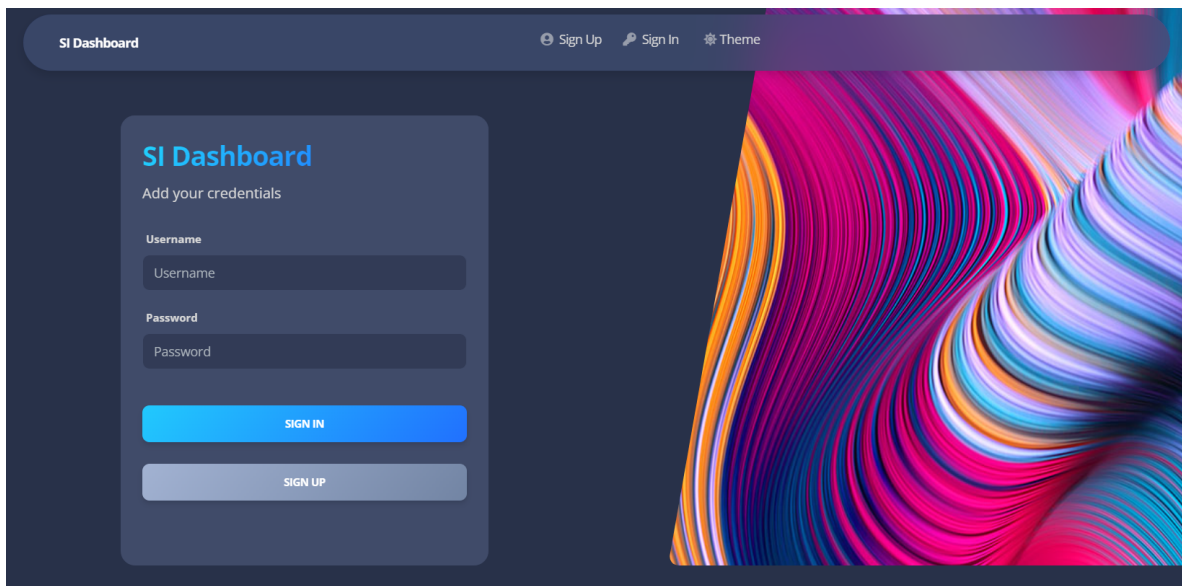


Figura 1: Login SI Dashboard

2.1.1. Estructura del proyecto

El proyecto cuenta con la siguiente estructura:

```
├── CHANGELOG.md
├── Dockerfile
├── LICENSE.md
├── README.md
├── api_generator
│   ├── commands.py
│   ├── forms
│   ├── manager.py
│   └── routes
├── apps
│   ├── __init__.py
│   ├── api
│   ├── authentication
│   ├── config.py
│   ├── db.sqlite3
│   ├── home
│   ├── models.py
│   ├── static
│   └── templates
├── build.sh
├── docker-compose.yml
├── env.sample
├── gunicorn-cfg.py
├── nginx
│   └── appseed-app.conf
├── package.json
├── render.yaml
├── requirements.txt
├── run.py
├── tratamiento
│   ├── Ejercicio1.drawio
│   ├── Memoria.pdf
│   ├── __init__.py
│   ├── database
│   ├── dependencies.sh
│   └── src
└── 13 directories, 23 files
```

Figura 2: Estructura proyecto

- Dockerfile, docker-compose.yml, gunicorn-cfg.py, render.yaml, etc.
- **api_generator** el cual genera bloques de texto para includes en HTML
- **apps** contiene toda la información sobre el backend de la aplicación (*.html, .css, .py Flask backend, .js, db.sqlite3 (contiene información sobre los usuarios)*)
- **nginx** contiene el archivo de configuración de nginx
- **tratamiento** incluye los archivos de la *práctica 1*

3. Ejercicio 1

Para simular el CMI, hemos utilizado una librería en Python: *Flask*

Hemos creado una página *index.html* la cual muestra los gráficos, junto a las opciones de selección para cada uno de los gráficos

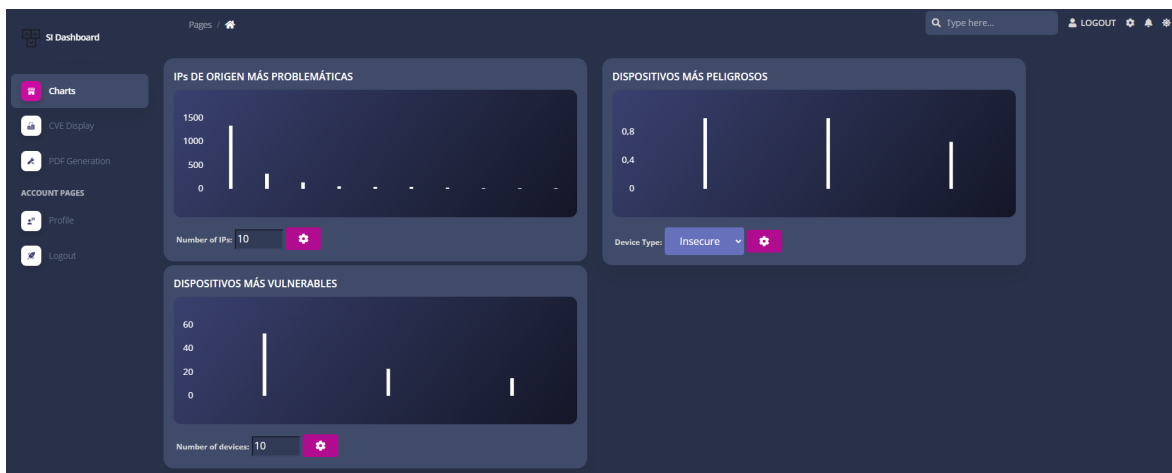


Figura 3: Página de index

Como se ve en la imagen, contamos con unos botones los cuales nos permiten modificar la información de los gráficos.

Como podemos comprobar en la imagen 4, al seleccionar el número de IPs = 4, únicamente nos muestra 4 IPs. Mientras que al seleccionar número de dispositivos = 3 como en la imagen 3 únicamente nos muestra 3 dispositivos.

Pese a que se realicen los cambios sobre los gráficos, el número de selección siempre se pone a los valores por defecto

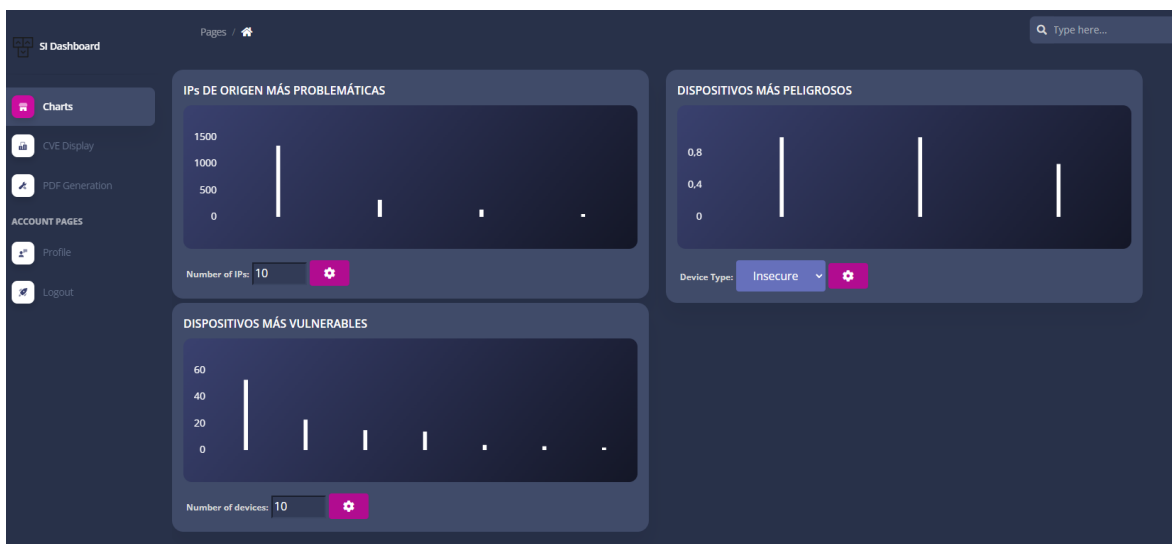


Figura 4: Graficos Modificados

Tras añadir varios datos, hemos decidido establecer un orden y una unificación en nuestro CMI. Donde dividimos nuestros gráficos gráficos IPs, de dispositivos y en función de fechas:

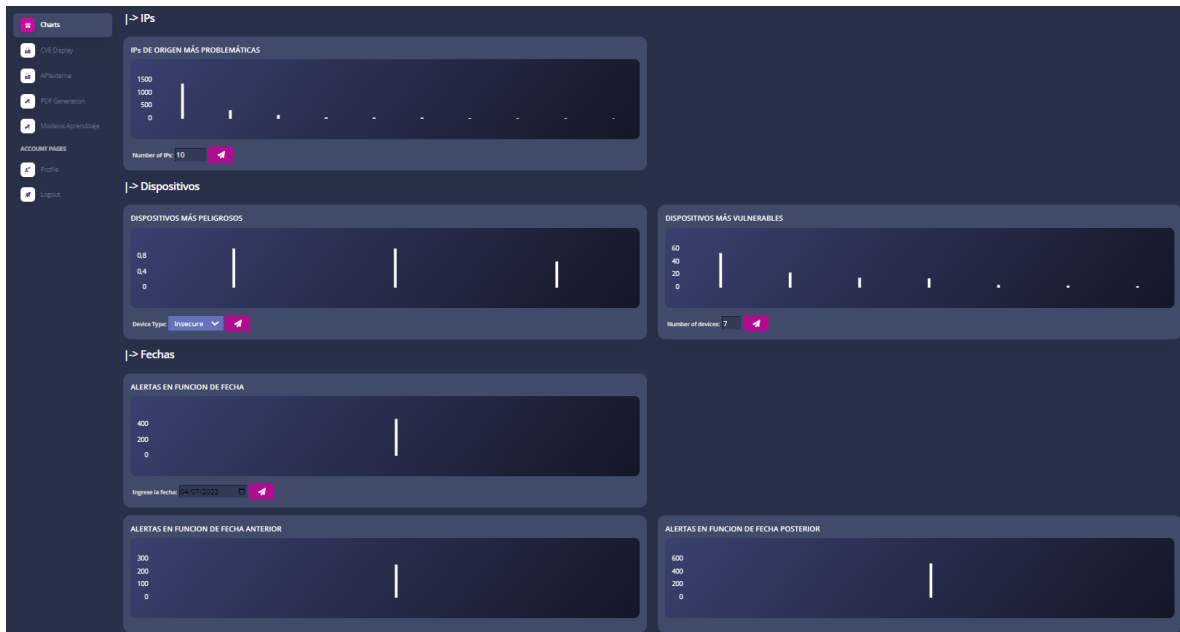


Figura 5: CMI Ordenado

4. Ejercicio2

En este ejercicio se nos pide la creación de una nueva métrica, la cual nos muestre los dispositivos más peligrosos, en función del número de servicios inseguros frente al total de servicios.

Nuestro archivo `get_most_dangerous.py` dentro de la carpeta de `tratamiento/src/graficos` realiza las queries a la BD generada durante la práctica 1:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import sqlite3
4 import json
5
6 def get_most_dangerous(code):
7     # Connect to the SQLite3 database file
8     conn = sqlite3.connect('/tratamiento/database/base.db')
9     # Querys to "devices" table & "alerts" table
10    query_analisis = 'SELECT ip, servicios, servicios_vulnerables FROM analisis'
11    # Read the querys into the dataframe
12    df_analisis = pd.read_sql_query(query_analisis, conn)
13    df_analisis.rename(columns = {'ip': 'IP'}, inplace= True)
14    df_analisis["porcentaje_inseguro"] = df_analisis["servicios_vulnerables"]/df_analisis["servicios"]
15    df_seguros = df_analisis[df_analisis["porcentaje_inseguro"] <= 0.33]
16    df_inseguros = df_analisis[df_analisis["porcentaje_inseguro"] > 0.33]
17
18    if (code == 0):
19        chart_dict = {
20            "labels": df_inseguros['IP'].tolist(),
21            "datasets": [{
22                "label": "porcentaje_inseguro",
23                "backgroundColor": "white",
24                "borderColor": "white",
25                "borderWidth": 3,
26                "data": df_inseguros['porcentaje_inseguro'].tolist(),
27                "fill": True,
28                "maxBarThickness": 6
29            }]
30        }
31    else:
32        chart_dict = {
33            "labels": df_seguros['IP'].tolist(),
34            "datasets": [{
35                "label": "porcentaje_inseguro",
36                "backgroundColor": "white",
37                "borderColor": "white",
38                "borderWidth": 3,
39                "data": df_seguros['porcentaje_inseguro'].tolist(),
40                "fill": True,
41                "maxBarThickness": 6
42            }]
43        }
44
45    # Convert the dictionary to JSON format
46    chart_json = json.dumps(chart_dict)
47
48    print(chart_json)
49    return chart_json
50
```


Como se muestra en las imágenesn 3 y 4, el gráfico **"Dispositivos más peligrosos"** contiene la información relativa a la query realizada por la función.



Figura 6: Gráfico Dispositivos Peligrosos

5. Ejercicio 3

En este ejercicio se propone la implementación de la consulta de los últimos 10 CVE a la api <https://www.cve-search.org/api/last>

Para ello, hemos implementado la siguiente función en el backend 7

```
@blueprint.route('/cve')
@login_required
def CVE():

    response = requests.get('https://cve.circl.lu/api/last')

    if response.status_code == 200:
        vulnerabilities = response.json()[:10]

        print(vulnerabilities)
    else:
        vulnerabilities = [{'error': 'Error al obtener las vulnerabilidades'}]
        print(vulnerabilities)

    data = json.dumps(vulnerabilities)

    return render_template('home/cve.html', data=data)
```

Figura 7: Función para obtener los CVE

Esto renderiza la template *cve.html* devolviendo:

```
{
  "Modified": "2023-05-02T15:59:00",
  "Published": "2023-04-21T16:15:00",
  "access": {},
  "assigner": "305.Information-Security@3ds.com",
  "capes": [
    {
      "id": "591",
      "name": "Reflected XSS",
      "prerequisites": "An application that leverages a client-side web browser with scripting enabled. An application that fail to adequately sanitize or encode untrusted input.",
      "related_weakness": [
        "79"
      ],
      "solutions": "Use browser technologies that do not allow client-side scripting. Utilize strict type, character, and encoding enforcement. Ensure that all user-supplied input is validated before",
      "summary": "This type of attack is a form of Cross-Site Scripting (XSS) where a malicious script is \"reflected\" off a vulnerable web application and then executed by a victim's browser. The pr"
    },
    {
      "id": "209",
      "name": "XSS Using MIME Type Mismatch",
      "prerequisites": "The victim must follow a crafted link that references a scripting file that is mis-typed as a non-executable file. The victim's browser must detect the true type of a mis-label",
      "related_weakness": [
        "20",
        "646",
        "79"
      ],
      "solutions": "",
      "summary": "An adversary creates a file with scripting content but where the specified MIME type of the file is such that scripting is not expected. The adversary tricks the victim into accessin"
    },
    {
      "id": "588",
      "name": "DOM-Based XSS",
      "prerequisites": "An application that leverages a client-side web browser with scripting enabled. An application that manipulates the DOM via client-side scripting. An application that fails to",
      "related_weakness": [
        "20",
        "79",
        "83"
      ],
      "solutions": "Use browser technologies that do not allow client-side scripting. Utilize proper character encoding for all output produced within client-site scripts manipulating the DOM. Ensure",
      "summary": "This type of attack is a form of Cross-Site Scripting (XSS) where a malicious script is inserted into the client-side HTML being parsed by a web browser. Content served by a vulnerab"
    },
    {
      "id": "592",
      "name": "Stored XSS",
      "prerequisites": "An application that leverages a client-side web browser with scripting enabled. An application that fails to adequately sanitize or encode untrusted input. An application that",
      "related_weakness": [
        "79"
      ]
    }
  ]
}
```

Figura 8: CVE

6. Ejercicio 4

En este ejercicio "libre" se proponen una serie de implementaciones para nuestro CMI

6.1. Login Page

En nuestro CMI contamos con un sistema de registro e identificador de usuarios persistente mediante el gestor de BBDD *sqlite3*. Nuestra página de login es la siguiente: 9

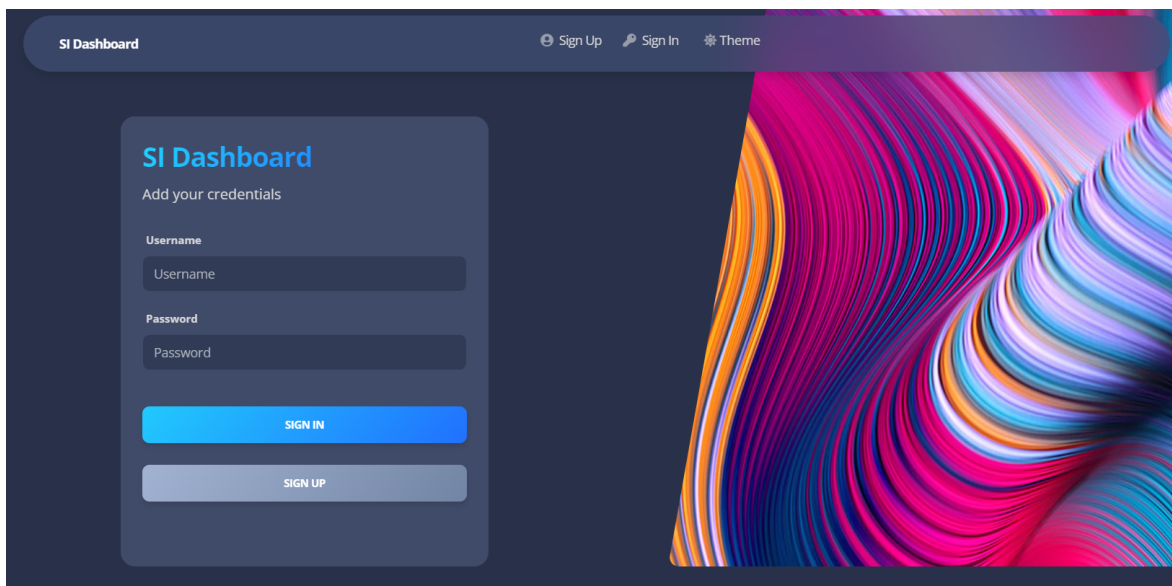


Figura 9: Login page

6.2. Generación PDF y análisis de otras métricas

En nuestro CMI hemos optado por combinar estos dos aspectos. Para ello, hemos creado una página *pdf.html*, la cual contiene únicamente un formulario de selección de métrica para descargar un PDF.

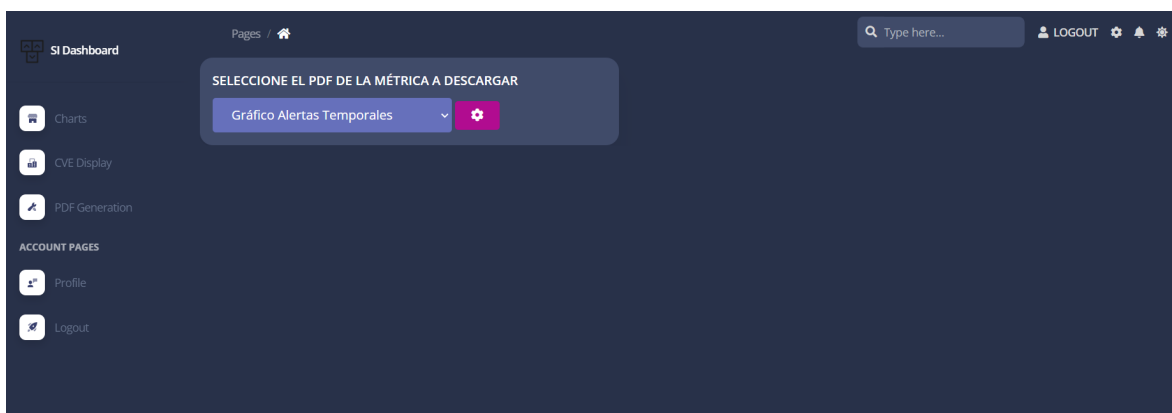


Figura 10: PDF Generation

La métricas proporcionadas para su descarga en PDF son las correspondientes métricas creadas durante el desarrollo de la práctica 1: *Alertas Temporales*, *Tipo de Alertas*, *Dispositivos más vulnerables*, *Puertos más vulnerados*, *IPs origen más peligrosas*

6.3. Analisis de otras métricas

En cuanto al análisis de otras métricas, hemos optado por analizar el número de ataques se reciben en una fecha en comparación con su anterior y su sucesor. Es decir, que si queremos saber cuantas alertas obtenemos en la fecha 2022-07-04, podemos enviárselo a nuestro CMI y obtener las alertas del 2022-07-03 y del 2022-07-05. Con esta métrica podemos compaar muy de cerca las alertas en función del tiempo.

Para ello utilizamos nuestro archivos `get_alertas_temporal.py`, `get_alertas_temporal_next.py` y `get_alertas_temporal_prev.py`. Ubicados dentro de la carpeta `Tratamiento/src/Graficos`:

```
1 import json
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import sqlite3
5 from io import BytesIO
6
7
8 def get_alertas_temporal(date):
9     # Connect to the SQLite3 database file
10    conn = sqlite3.connect('/tratamiento/database/base.db')
11
12    # Querys to "alerts" table
13    query_alert = "SELECT * FROM alerts WHERE DATE(timestamp) = ?"
14
15    # Read the query into a df
16    df_alerts = pd.read_sql_query(query_alert, conn, params=[date])
17
18    # Change dates to date time
19    df_alerts['timestamp'] = pd.to_datetime(df_alerts['timestamp'])
20
21    # Set the index as the timestamp
22    df_alerts = df_alerts.set_index('timestamp')
23
24    # Group x day and count
25    alerts_per_day = df_alerts['prioridad'].resample('D').count()
26
27    chart_dict = {
28        "labels": alerts_per_day.index.strftime('%Y-%m-%d').tolist(),
29        "datasets": [{
30            "label": "Numero de alertas",
31            "backgroundColor": "white",
32            "borderColor": "white",
33            "borderWidth": 3,
34            "data": alerts_per_day.tolist(),
35            "fill": True,
36            "maxBarThickness": 6
37        }]
38    }
39
40    # Convert the dictionary to JSON format
41    chart_json = json.dumps(chart_dict)
42
43    print(chart_json)
44    return chart_json
45
```

Obteniendo así los datos en nuestro CMI, pudiendo realizar consultas sobre cualquier fecha almacenada:



Figura 11: Alertas 2022-07-04

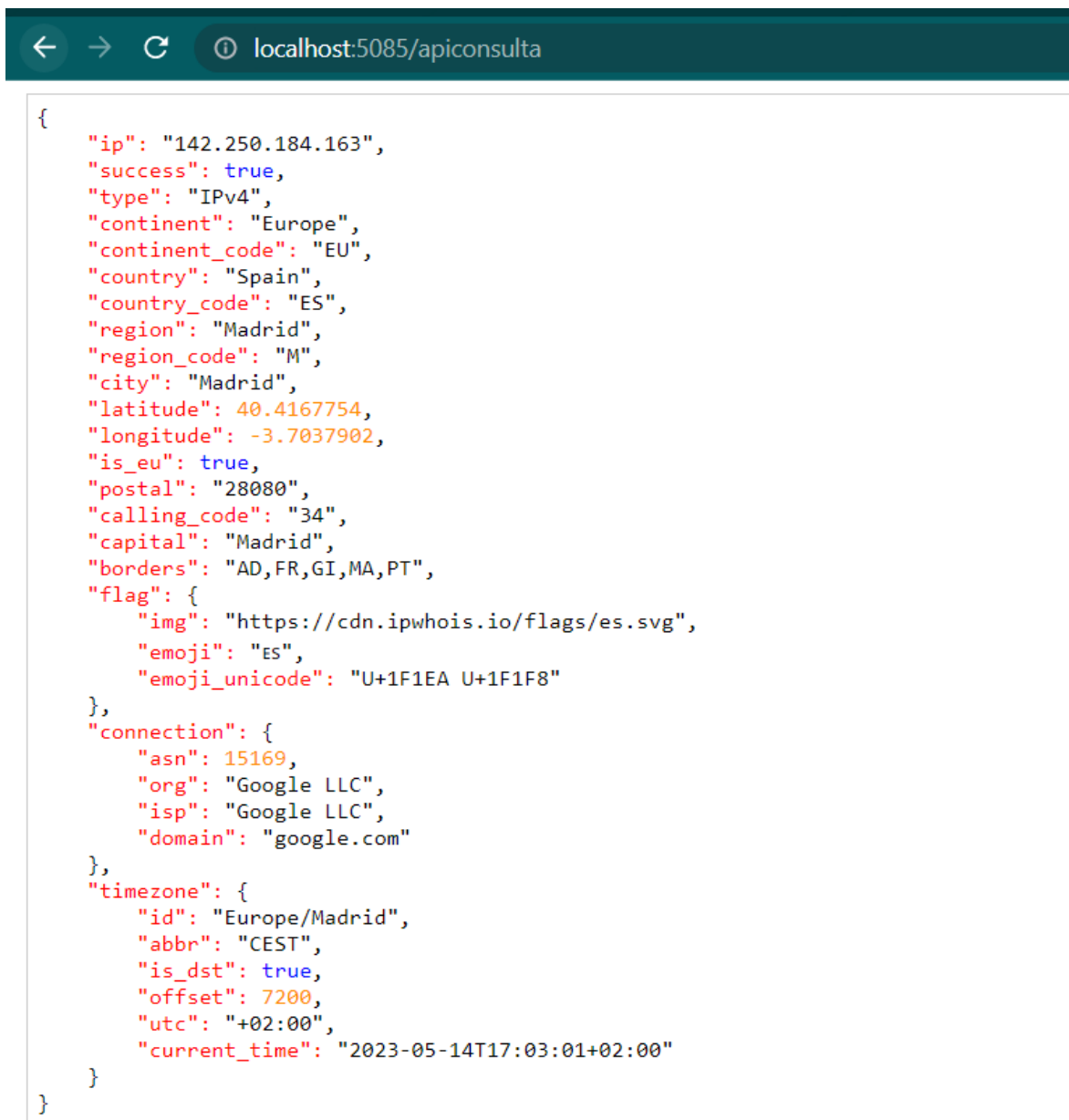
6.4. Mostrar datos de algun servicio web mediante otra API

Para la API externa hemos optado por una API que permite hacer consultas sobre una IP, devolviendo datos de geolocalización, proveedor, etc.

La petición a la API recibe como respuesta un JSON, que devolvemos lo más formateado posible, para hacer la petición a la API usamos los siguientes códigos en el backend.

```
1 @blueprint.route('/apiconsulta',methods=['POST'])
2 @login_required
3 def apiconsulta():
4
5
6     ip=request.form['ip']
7     peticion="http://ipwho.is/"
8     peticion=peticion+ip
9     response= requests.get(peticion)
10    if response.status_code == 200:
11        localizacion = response.json()
12
13        print(localizacion)
14    else:
15        localizacion = [{'error': 'Error al obtener la ip'}]
16        print(localizacion)
17
18    data = json.dumps(localizacion)
19
20    return render_template('home/cve.html', data=data)
```

Pasamos la IP a través de un POST, si la respuesta es un ok devolvemos la template preparada para formatear JSON, si no, devolvemos error: Un ejemplo de uso sería pasándole la IP 142.250.184.163:



```
{
  "ip": "142.250.184.163",
  "success": true,
  "type": "IPv4",
  "continent": "Europe",
  "continent_code": "EU",
  "country": "Spain",
  "country_code": "ES",
  "region": "Madrid",
  "region_code": "M",
  "city": "Madrid",
  "latitude": 40.4167754,
  "longitude": -3.7037902,
  "is_eu": true,
  "postal": "28080",
  "calling_code": "34",
  "capital": "Madrid",
  "borders": "AD,FR,GI,MA,PT",
  "flag": {
    "img": "https://cdn.ipwhois.io/flags/es.svg",
    "emoji": "ES",
    "emoji_unicode": "U+1F1EA U+1F1F8"
  },
  "connection": {
    "asn": 15169,
    "org": "Google LLC",
    "isp": "Google LLC",
    "domain": "google.com"
  },
  "timezone": {
    "id": "Europe/Madrid",
    "abbr": "CEST",
    "is_dst": true,
    "offset": 7200,
    "utc": "+02:00",
    "current_time": "2023-05-14T17:03:01+02:00"
  }
}
```

Figura 12: IP 142.250.184.163

7. Ejercicio 5

7.1. Realizar un método de Regresion Lineal

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from sklearn import datasets, linear_model
4 from sklearn.metrics import mean_squared_error, r2_score
5 import json
6
7 with open("dump/devices_IA_clases.json", "r") as f:
8     trainDevices = json.load(f)
9
10 with open("dump/devices_IA_predecir_v2.json", "r") as f:
11     testDevices = json.load(f)
12
13 xTrain = []
14 yTrain = []
15 xTest = []
16 yTest = []
17 yPredict = []
18 for i in trainDevices:
19     if i['servicios'] == 0:
20         xTrain.append([0])
21     else:
22         xTrain.append([i['servicios_inseguros']/i['servicios']])
23         yTrain.append([i['peligroso']])
24
25 for i in testDevices:
26     if i['servicios'] == 0:
27         xTest.append([0])
28     else:
29         xTest.append([i['servicios_inseguros'] / i['servicios']])
30         yTest.append([i['peligroso']])
31
32
33 regr = linear_model.LinearRegression()
34 regr.fit(xTrain,yTrain)
35
36 yPredict = regr.predict(xTest)
37 print("Mean squared error: %.2f" % mean_squared_error(yTest, yPredict))
38 plt.scatter(xTest, yTest, color="black")
39 plt.plot(xTest, yPredict, color="blue", linewidth=3)
40 plt.xticks(())
41 plt.yticks(())
42 plt.show()
```

En este modelo de regresión lineal cargamos los dos Json, tanto training como testing, y generamos el modelo, para posteriormente ponerlo a prueba con el set de testing. Como resultado de este modelo obtenemos la siguiente gráfica.

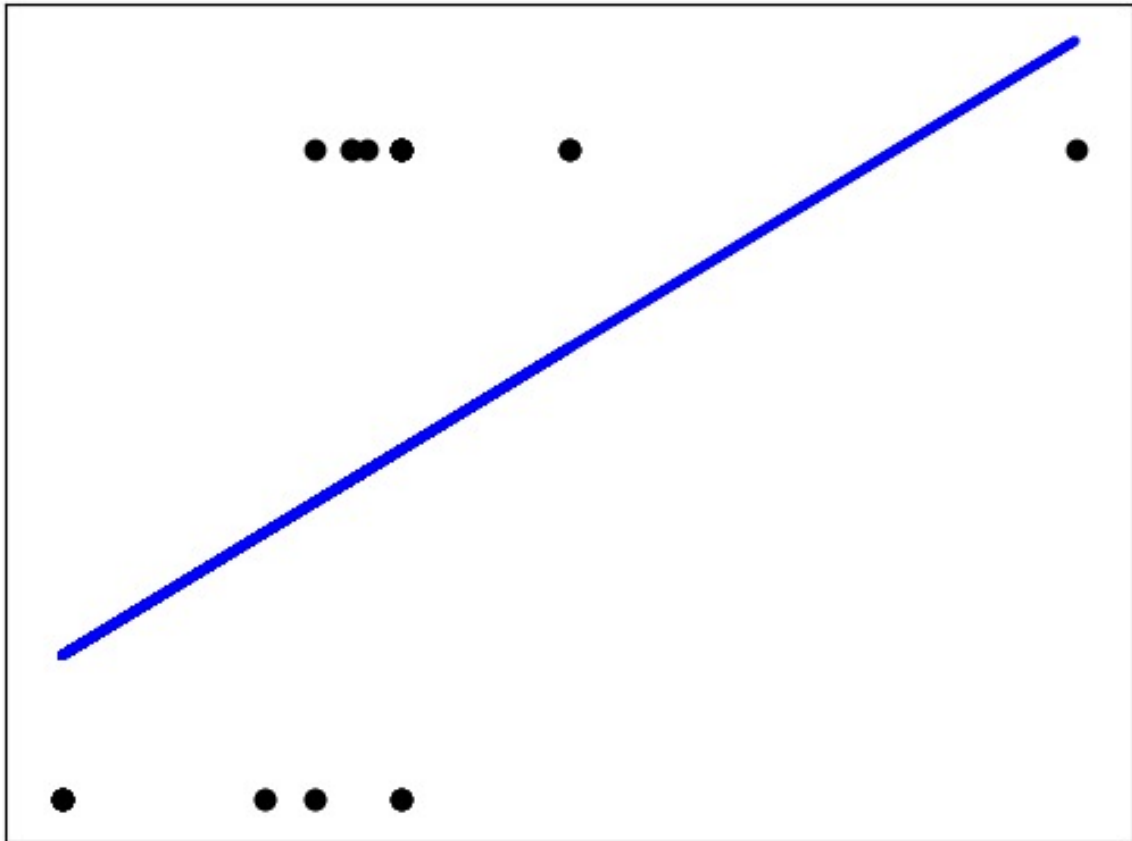


Figura 13: Modelo Regresión Linear

Esta gráfica puede consultarse en el apartado destinado a los modelos de aprendizaje supervisado dentro del CMI.

7.2. Realizar un método de Decision Tree

Para la realización de nuestro árbol de decisión con los datos a entrenar, debemos, primero cargarnos los datos con las siguientes funciones:

```
1
2 def cargar_datos_entrenamiento(archivo_json,X,y):
3     with open(archivo_json, 'r') as f:
4         datos = json.load(f)
5
6     for dispositivo in datos:
7         X.append([
8             dispositivo['servicios'],
9             dispositivo['servicios_inseguros'],
10        ])
11        y.append(dispositivo['peligroso'])
12
13 def predecir_datos(archivo_json,clf,Xp):
14     with open(archivo_json, 'r') as f:
15         datos = json.load(f)
16
17     for dispositivo in datos:
18         Xp.append([
19             dispositivo['servicios'],
20             dispositivo['servicios_inseguros']
21        ])
22     Yp = clf.predict(Xp)
23     device_ids = []
24     for device in datos:
25         device_ids.append(device['id'])
26     return Yp, device_ids
```

Ahora, nuestra funcion main se verá de la siguiente forma:

```
1
2 if __name__ == '__main__':
3     Xt = []
4     yt = []
5     Xp = []
6     Yp = []
7     devices_id = []
8     jsonTrain = 'dump/devices_IA_clases.json'
9     jsonPred = 'dump/devices_IA_predecir_v2.json'
10
11     # Load data
12     cargar_datos_entrenamiento(jsonTrain, Xt, yt)
13     clf = tree.DecisionTreeClassifier()
14     clf = clf.fit(Xt, yt)
15     # Print plot
16     dot_data = tree.export_graphviz(clf, out_file=None,
17                                     feature_names=['servicios', 'servicios_inseguros'],
18                                     class_names=['No peligroso', 'Peligroso'],
19                                     filled=True, rounded=True,
20                                     special_characters=True)
21     graph = graphviz.Source(dot_data)
22     graph.view()
23
24     Yp, devices_id = predecir_datos(jsonPred,clf,Xp)
25
```

```

26 peligrosos = []
27 for i in range(len(Yp)):
28     if Yp[i] == 1:
29         peligrosos.append(devices_id[i])
30
31 print(Yp)
32 print(peligrosos)

```

Este código, carga los datos de entrenamiento, genera con estos un árbol de decisión y lo printea. Seguidamente, carga los datos a predecir y en función de lo anterior, devuelve una lista con los dispositivos a predecir que consideramos "peligrosos":

```

Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)]
[1 1 1 0 0 1 1 1 1 0 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 1 0]
['proxy1', 'webserver1', 'pc_luis', 'webserver2', 'pc_jose', 'server1', 'proxy3',

```

Figura 14: Dispositivos Peligrosos

El árbol de decisión generado lo podemos ver en nuestro CMI en Modelos de Aprendizaje:

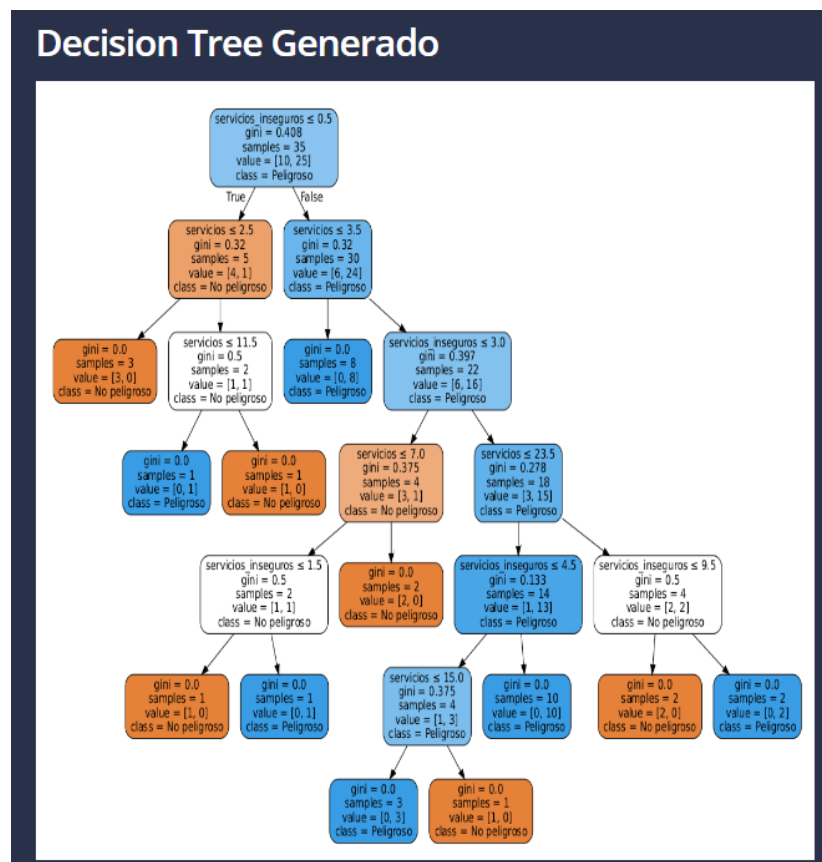


Figura 15: Decision Tree

7.3. Realizar un método de Random forest

Para la realización de nuestro random forest con los datos a entrenar, debemos, primero cargarnos los datos con las siguientes funciones:

```
1
2 def cargar_datos_entrenamiento(archivo_json,X,y):
3     with open(archivo_json, 'r') as f:
4         datos = json.load(f)
5
6     for dispositivo in datos:
7         X.append([
8             dispositivo['servicios'],
9             dispositivo['servicios_inseguros'],
10        ])
11        y.append(dispositivo['peligroso'])
12
13 def predecir_datos(archivo_json,clf,Xp):
14     with open(archivo_json, 'r') as f:
15         datos = json.load(f)
16
17     for dispositivo in datos:
18         Xp.append([
19             dispositivo['servicios'],
20             dispositivo['servicios_inseguros']
21        ])
22     Yp = clf.predict(Xp)
23     device_ids = []
24     for device in datos:
25         device_ids.append(device['id'])
26     return Yp, device_ids
```

Ahora, nuestra funcion main se verá de la siguiente forma:

```
1
2 if __name__ == '__main__':
3     Xt = []
4     yt = []
5     Xp = []
6     Yp = []
7
8     jsonTrain = 'dump/devices_IA_clases.json'
9     jsonPred = 'dump/devices_IA_predecir_v2.json'
10    cargar_datos_entrenamiento(jsonTrain, Xt, yt)
11    clf = RandomForestClassifier(max_depth=2, random_state=0,n_estimators=10)
12    clf.fit(Xt, yt)
13    print(clf.predict(Xt))
14
15    for i in range(len(clf.estimators_)):
16        print(i)
17        estimator = clf.estimators_[i]
18        export_graphviz(estimator,
19                        out_file='tree.dot',
20                        feature_names=['servicios', 'servicios_inseguros'],
21                        class_names=['No peligroso', 'Peligroso'],
22                        rounded=True, proportion=False,
23                        precision=2, filled=True)
24        call(['dot', '-Tpng', 'tree.dot', '-o', 'tree'+str(i)+'.png', '-Gdpi=600'])
```

