CUNEF

Máster Universitario en Ciencia de Datos

Aprendizaje automático

Productivización del modelo

Diego Cendan
Aitor Larrinoa

Enero 2021

Índice

1. Introducción

El presente trabajo muestra la puesta en producción de un modelo de Aprendizaje automático. Más concretamente se trata del modelo Random Forest calculado en la anterior práctica del curso. Para ello, se hará uso del módulo de Python *Flask*. Este módulo nos permite crear aplicaciones web de forma rápida y sencilla. Además, también haremos uso de contenedores Docker. Estos contenedores nos permitirán poder hacer uso de nuestra aplicación en cualquier máquina del mundo, evitando así el gran problema *"En mi ordenador si funciona"*.

En la práctica se han planteado dos formas diferentes de crear esta aplicación web. A continuación explicamos cada una de ellas:

- *Forma 1*: La primera de las formas es la menos 'interactiva" para el usuario. El usuario deberá ejecutar la aplicación y deberá introducir la url indicada por la aplicación en Postman. Una vez en Postman, deberá introducir los datos en formato JSON y deberá clicar en el botón *SEND*. Una vez haga eso, aparecerá la predicción de nuestro modelo.
- *Forma 2*: En la segunda forma lo que hemos hecho ha sido crear una aplicación mediante html. Es decir, cuando se ejecute nuestra aplicación, se abrirá una página en la web en la que el usuario introducirá los datos en cada una de las casillas que aparecen. Después, el propio usuario dispondrá de un botón con el que poder predecir con los datos introducidos. Una vez presionado el botón, aparecerá una nueva ventana en la que el usuario podrá visualizar si habrá o no fallecidos en función de los datos introducidos.

2. Forma 1

2.1. Flask

Para que nuestra aplicación Flask pueda funcionar, necesita de una aplicación que realice todos los cálculos necesarios. Esta aplicación será un fichero de Python y nosotros lo definiremos bajo el nombre de *app.py*. A continuación mostremos el código de la aplicación separado en diferentes bloques, así posteriormente podremos explicar qué ocurre en cada uno de los bloques.

```
import pandas as pd
import pickle
import math
import numpy as np
from flask import Flask,  request, jsonify
from sklearn.metrics import roc_curve
```

```
app = Flask(__name__)
```

```
pickle.load(open('RF.pkl', 'rb'))

@app.route('/prediction',methods=["POST"])

def predict():
    json_ = request.json
    data_dictionary=dict()
    for variable in json_:
        if variable == "C_MNTH":
            month = int(json_[variable])
            NA_cos_C_MNTH = np.cos(2*math.pi*month/12)
            NA_sin_C_MNTH = np.sin(2*math.pi*month/12)
            data_dictionary["NA_cos_C_MNTH"] = NA_cos_C_MNTH
            data_dictionary["NA_sin_C_MNTH"] = NA_sin_C_MNTH
        elif variable == "C_HOUR":
            hour = int(json_[variable])
            NA_cos_C_HOUR = np.cos(2*math.pi*hour/24)
            NA_sin_C_HOUR = np.sin(2*math.pi*hour/24)
            data_dictionary["NA_cos_C_HOUR"] = NA_cos_C_HOUR
            data_dictionary["NA_sin_C_HOUR"] = NA_sin_C_HOUR
        elif variable == "C_WDAY":
            wday = int(json_[variable])
            NA_cos_C_WDAY = np.cos(2*math.pi*wday/7)
            NA_sin_C_WDAY = np.sin(2*math.pi*wday/7)
            data_dictionary["NA_cos_C_WDAY"] = NA_cos_C_WDAY
            data_dictionary["NA_sin_C_WDAY"] = NA_sin_C_WDAY
        elif variable == "num_C_YEAR":
            value_c_year = int(json_[variable])
            mean_c_year_test = 2006.0155218348198
            sdv_c_year_test = 4.567695934222996
            new_value_year = (value_c_year - mean_c_year_test)/
               sdv_c_year_test
            data_dictionary["num_C_YEAR"] = new_value_year
        elif variable == "num_Random":
            value_Random = int(json_[variable])
            mean_Random_test = 49.51538088565281
            sdv_Random_test = 28.850311918400497
```

```
            new_value_Random = (value_Random-mean_Random_test)/
                sdv_Random_test
            data_dictionary["num_Random"] = new_value_Random
        elif variable == "num_C_VEHS":
            value_c_vehs = int(json_[variable])
            mean_c_vehs_test = 2.0971652406757
            sdv_c_vehs_test = 1.3324953572064602
            new_value_vehs = (value_c_vehs-mean_c_vehs_test)/
                sdv_c_vehs_test
            data_dictionary["num_C_VEHS"] = new_value_vehs
        else:
            data_dictionary[variable] = json_[variable]
    query_df = pd.DataFrame(data_dictionary, index = [0])
    prediction = model.predict(query_df)
    prediction_proba = model.predict_proba(query_df)
    predictions_new=(prediction_proba[:,1] >= 0.9375).astype(int)
    return jsonify({"prediction":int(predictions_new[0])})
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

- *Bloque 1*: En el primer bloque cargamos las librerías necesarias para que nuestra aplicación funcione.
- *Bloque 2*: Generamos una aplicación creando una instancia usando el método flask.Flask(__name__). Se trata de una forma conveniente de obtener el nombre de importación del lugar donde se define la aplicación. Flask usa el nombre de importación para saber dónde buscar recursos, plantillas, archivos estáticos, carpetas de instancias, etc.
- *Bloque 3*: En el tercer bloque cargamos nuestro modelo en formato pickle, perviamente entrenado en el notebook *03_Modelos*.
- *Bloque 4*: El cuarto bloque es el que contiene el *core* de la aplicación. En él obtenemos los datos del json, introducido por el usuario, y los modificamos hasta obtener un diccionario en el que las claves son las variables y los valores son los valores que introduce el usuario después de modificarlos. En cuanto a las modificaciones realizadas, las variables temporales deben tener componente coseno y componente seno, pues así ha sido entrenado nuestro modelo. Por otro lado, las variables numéricas deben ser estandarizadas, es por ello que hemos tomado los datos de la media y desviación típica de nuestras variables numéricas en el datset de testing y las hemos introducido manualmente para poder calcular la estandarización de las variables numéricas introducidas por el usuario. Finalmente, dado que las variables one hot son introducidas por el usuario en formato adecuado, no deberemos realizar ninguna modificación. En la función predict, al final, hacemos uso del threshold óptimo obtenido por la curva roc en el notebook *03_modelos* para calcular nuestro nuevo vector de predicción y finalmente devolver el valor de la predicción.
- *Bloque 5*: En el último bloque hemos añadido la condición if __name__ == "__main__". Cada módulo Python tiene _name_ definido, y en el caso de ser "_main_" implica que el usuario está ejecutando el módulo de forma independiente. Es decir, se utiliza para ejecutar aquel código en el que el archivo se ejecuta directamente sin ser importado

previamente. App.run ejecutará el servidor en desarrollo habilitando el método de depuración del navegador debug=True con el fin de devolver únicamente la predicción.

Una vez definida nuestra aplicación, podemos comenzar a probarla. Para ello utilizaremos una herramienta llamada *Postman*.

2.2. Postman

Postman es una herramienta nos permite crear peticiones sobre APIs de una forma muy sencilla y poder, de esta manera, probar las APIs. Luego, en nuestro caso, haremos uso de *Postman* para mandar a nuestra aplicación un JSON, con la información de un accidente, y que ésta nos devuelva la predicción.

Para probar que todo funciona correctamente, realizaremos dos peticiones a nuestra API. Una que nos deberá devolver *"habrá al menos un fallecido"* y otra que nos deberá devolver *"no habrá fallecido"*.

2.2.1. Petición 1

En esta primera petición esperamos que la predicción nos devuelva *"habrá al menos un fallecido"*. Para confirmar que ocurra esto, elegiremos unos datos concretos que sabemos que nos devolverán como resultado la predicción buscada. Los datos que utilizaremos son:

- *Mes del accidente*: Enero.
- *Día de la semana*: Miércoles.
- *Hora*: Las 10:00.
- *Configuración del accidente*: Dos vehículos en dirección diferente.
- *Configuración de la carretera*: Sin intersección
- *Clima*: Clima desfavorable.
- *Condiciones de la carretera*: Desfavorable, condiciones adversas.
- *Tipo de carretera*: Carretera recta, con curvas.
- *ontrol de tráfico*: Las señales de tráfico funcionan correctamente.
- *Año del accidente*: 2022.
- *Número de vehículos involucrados*: 2.
- *Valor random*: 20.

Luego, si introducimos estos datos en formato JSON en Postman, veamos cual es la predicción obtenida:

Podemos comprobar como efectivamente, el modelo nos devuelve como predicción un 0, lo que quiere decir que al menos habrá un fallecido.

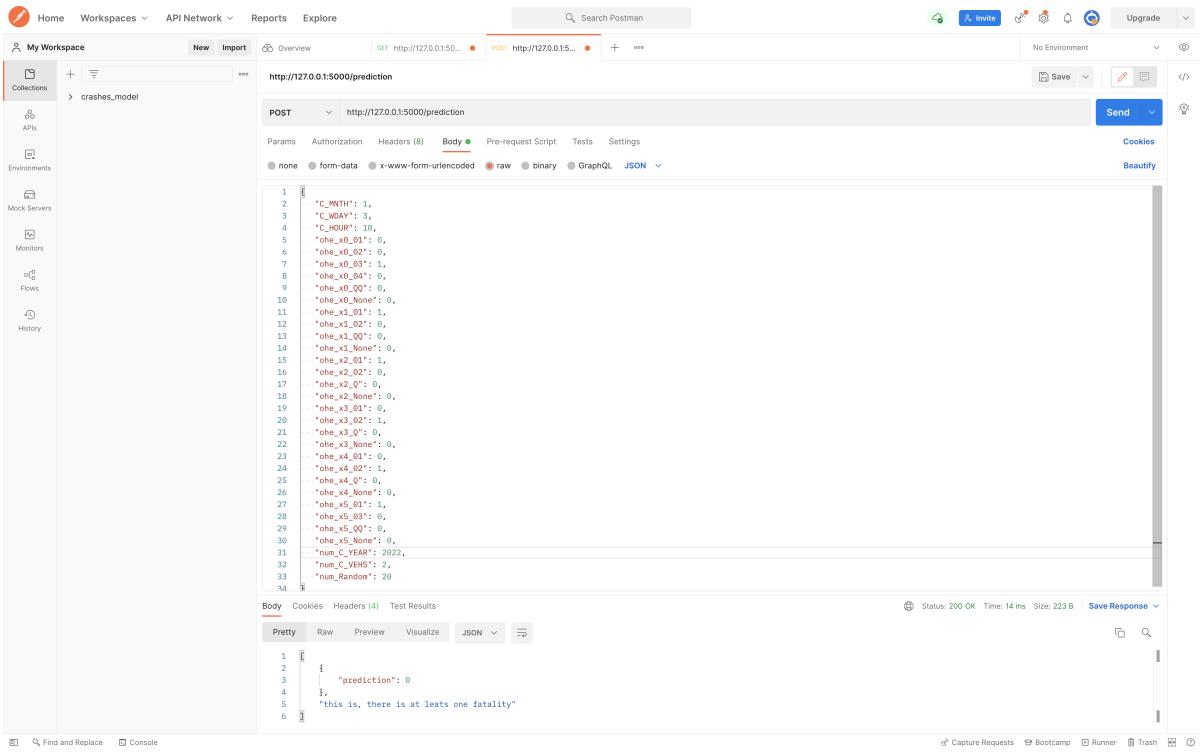

Figura 1: Resultado de la petición 1

2.2.2. Petición 2

En esta ocasión deberemos introducir diferentes datos en Postman para obtener la predicción *"no habrá fallecidos"*. Los datos de los que haremos uso serán los siguientes:

- *Mes del accidente*: Diciembre.
- *Día de la semana*: Lunes.
- *Hora*: Las 12:00.
- *Configuración del accidente*: Dos vehículos en la misma dirección.
- *Configuración de la carretera*: Sin intersección
- *Clima*: Clima desfavorable.
- *Condiciones de la carretera*: Desfavorable, condiciones adversas.
- *Tipo de carretera*: Carretera recta, con curvas.
- *Control de tráfico*: Las señales de tráfico funcionan correctamente.
- *Año del accidente*: 2021.
- *Número de vehículos involucrados*: 2.
- *Valor random*: 20.

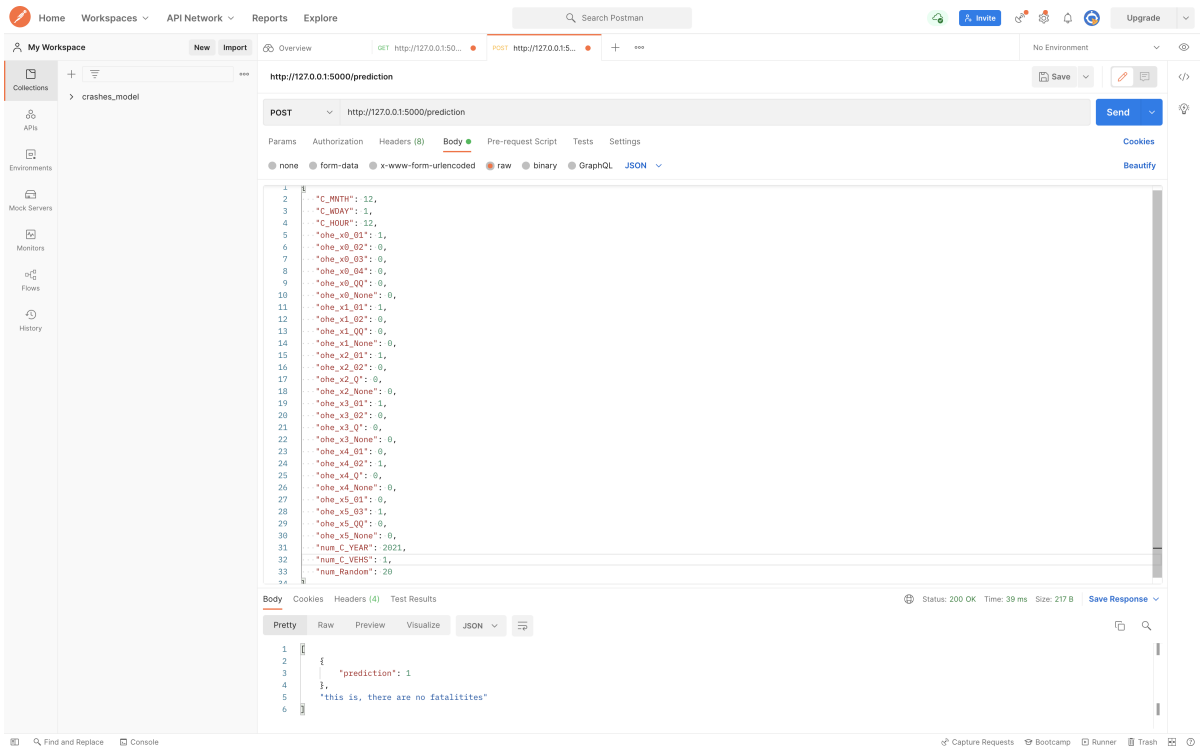

Figura 2: Resultado de la petición 2

Veamos ahora como introducimos nuestros datos, en formto JSON, y la respuesta que nos devuelve el Postman.

Y, como podemos observar, el resultado es que no hay fallecimientos en el accidente.

3. Forma 2

3.1. Flask

En esta segunda forma, tal y como se ha mencionado en la introducción, haremos uso de https para que el usuario pueda interactuar. Para ello necesitaremos un fichero con extensión .py que se encargará de que todo funcione correctamente y dos archivos http. Comencemos con nuestra aplicación flask a la que hemos llamado *app.py*.

A continuación mostramos el código coloreado por bloques. Posteriormente explicaremos cada uno de estos bloques por separado.

```
import pandas as pd
import pickle
import math
import numpy as np
from flask import Flask, render_template, request

variables = ['num_C_YEAR', 'C_MNTH', 'C_WDAY','C_HOUR', 'num_C_VEHS', '
    num_Random', 'ohe_x0_01', 'ohe_x0_02', 'ohe_x0_03', 'ohe_x0_04', '
    ohe_x0_QQ', 'ohe_x0_None', 'ohe_x1_01', 'ohe_x1_02','ohe_x1_QQ', '
```

```
    ohe_x1_None', 'ohe_x2_01', 'ohe_x2_02', 'ohe_x2_Q', 'ohe_x2_None', '
    ohe_x3_01', 'ohe_x3_02', 'ohe_x3_Q', 'ohe_x3_None', 'ohe_x4_01', '
    ohe_x4_02', 'ohe_x4_Q', 'ohe_x4_None', 'ohe_x5_01', 'ohe_x5_03', '
    ohe_x5_QQ', 'ohe_x5_None']

app = Flask(__name__)

try:
    model=pickle.load(open('Models/RF.pkl', 'rb'))
except:
    print("Ha ocurrido un error al cargar el modelo")

@app.route('/')
def home():
    return render_template("index.html")

@app.route('/predict/',methods=["GET", "POST"])
def modify_data():
    data_dictionary = dict()
    for variable in variables:
        if variable == "C_MNTH":
            month = int(request.form.get(variable))
            print(month)
            NA_cos_C_MNTH = np.cos(2*math.pi*month/12)
            NA_sin_C_MNTH = np.sin(2*math.pi*month/12)
            data_dictionary["NA_cos_C_MNTH"] = NA_cos_C_MNTH
            data_dictionary["NA_sin_C_MNTH"] = NA_sin_C_MNTH
        elif variable == "C_HOUR":
            hour = int(request.form.get(variable))
            print(hour)
            NA_cos_C_HOUR = np.cos(2*math.pi*hour/24)
            NA_sin_C_HOUR = np.sin(2*math.pi*hour/24)
            data_dictionary["NA_cos_C_HOUR"] = NA_cos_C_HOUR
            data_dictionary["NA_sin_C_HOUR"] = NA_sin_C_HOUR
        elif variable == "C_WDAY":
            wday = int(request.form.get(variable))
            print(wday)
            NA_cos_C_WDAY = np.cos(2*math.pi*wday/7)
            NA_sin_C_WDAY = np.sin(2*math.pi*wday/7)
            data_dictionary["NA_cos_C_WDAY"] = NA_cos_C_WDAY
            data_dictionary["NA_sin_C_WDAY"] = NA_sin_C_WDAY
        elif variable == "num_C_YEAR":
            value_c_year = int(request.form.get(variable))
            mean_c_year_test = 2006.0155218348198
            sdv_c_year_test = 4.567695934222996
            new_value_year = (value_c_year - mean_c_year_test)/
               sdv_c_year_test
            data_dictionary["num_C_YEAR"] = new_value_year
        elif variable == "num_Random":
            value_Random = int(request.form.get(variable))
            mean_Random_test = 49.51538088565281
            sdv_Random_test = 28.850311918400497
            new_value_Random = (value_Random-mean_Random_test)/
               sdv_Random_test
            data_dictionary["num_Random"] = new_value_Random
        elif variable == "num_C_VEHS":
```

```
            value_c_vehs = int(request.form.get(variable))
            mean_c_vehs_test = 2.0971652406757
            sdv_c_vehs_test = 1.3324953572064602
            new_value_vehs = (value_c_vehs-mean_c_vehs_test)/
               sdv_c_vehs_test
            data_dictionary["num_C_VEHS"] = new_value_vehs
        else:
            data_dictionary[variable] = request.form.get(variable)

    prediction = predict(data_dictionary)

    return render_template('predict.html', prediction = prediction)
```

```
def predict(dictionary, threshold_opt=0.9735):
    query_df = pd.DataFrame(dictionary, index = [0])
    predictions_proba = model.predict_proba(query_df)
    prediction=(predictions_proba[:,1] >= threshold_opt).astype(int)
    return prediction
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Expliquemos ahora bloque tras bloque el código que acabamos de mostrar.

- *Bloque 1*: En este primer bloque cargamos las librerías y funciones necesarias para el desarrollo de la aplicación. También definimos una variable llamada *variables*. *variables* es una lista que contiene el nombre de todas las variables que debemos introducir en el modelo.
- *Bloque 2*: En el segundo bloque creamos una aplicación Flask. Para hacer esto le pasamos *__name__* a la clase Flask. Lo que hacemos es asociar la variable *__name__* al nombre del módulo. Esto es, en nuestro caso, tenemos la aplicación app.py
- *Bloque 3*: En el tercer bloque cargamos nuestro modelo. Si el modelo no puede cargarse aparecerá un mensaje por pantalla diciendo que ha habido un problema al cargar el modelo. En este caso se trata de un Random Forest que previamente ha sido guardado bajo el nombre y la extensión *RF.pkl*.
- *Bloque 4*: Creamos una primera instancia @app.route para decirle a Flask qué URL debe activar nuestra función. Cuando el usuario acceda a la URL /", automáticamente se iniciará la función *home()*. Dicha función, se encargará de devolvernos un html, en nuestro caso, *index.html* situado en la carperta templates, que servirá para que el usuario introduzca los datos. La función *render_template()* se encarga de buscar el archivo deseado en nuestra carpeta de templates y genera un html a partir de dicho archivo. Más adelante mostraremos una imagen de la interfaz index.html.
- *Bloque 5*: El quinto bloque es el más grande de todos. Los bloques 5 y 6 entran dentro de *@app.route("/predict/")*. Esto quiere decir que cuando el usuario acceda a la url "/predict/", se accionará todo lo que se encuentre dentro del app.route. Hablemos ahora de la función del bloque 5. Esta función se encargará de tomar el input generado por el usuario, en el bloque anterior, y lo transformará de forma que el modelo pueda ser utilizado para predecir. Lo primero que hacemos es codificar las variables temporales con su respectiva componente trigonométrica. Después, tomamos las variables

numéricas *Año del vehículo*, *Valor random* y *Número de vehículos involucrados en el accidente* y estandariza los valores de acuerdo con las medias y desviaciones típicas de las variables calculadas en el notebook *02_Preparación_datos*. Todas estas variables modificadas son introducidas en un diccionario en el que las claves son los nombres de las variables y los valores son los valores transformados (si han necesitado algún tipo de transformación) de las variables.

- *Bloque 6*: El sexto bloque está dentro de la ruta */predict/* y se trata de la función predict(). Esta función toma como entrada un diccionario y el threshold óptimo proporcionado por la curva de roc. La función convierte el diccionario a un data frame y calcula el vector de probabilidades de dicho data frame. Después, con

- *Bloque 7*: En el último bloque hemos añadido la condición if __name__ == "__main__". Cada módulo Python tiene _name_ definido, y en el caso de ser "_main_" implica que el usuario está ejecutando el módulo de forma independiente. Es decir, se utiliza para ejecutar aquel código en el que el archivo se ejecuta directamente sin ser importado previamente. App.run ejecutará el servidor en desarrollo habilitando el método de depuración del navegador debug=True con el fin de devolver únicamente la predicción.

Hemos podido ver que nuestra aplicación hace uso de dos html, *index.html* y *predict.html*. El primero de ellos se refiere a la interfaz en la que el usuario introduce los datos y, el segundo se refiere a la interfaz en la que el usuario puede ver la predicción obtenida. Debido a que este curso no trata de desarrollar html, no vamos a mostrar el código html pero sí vamos a mostrar la imagen de la interfaz en la que el usuario debe introducir los datos.

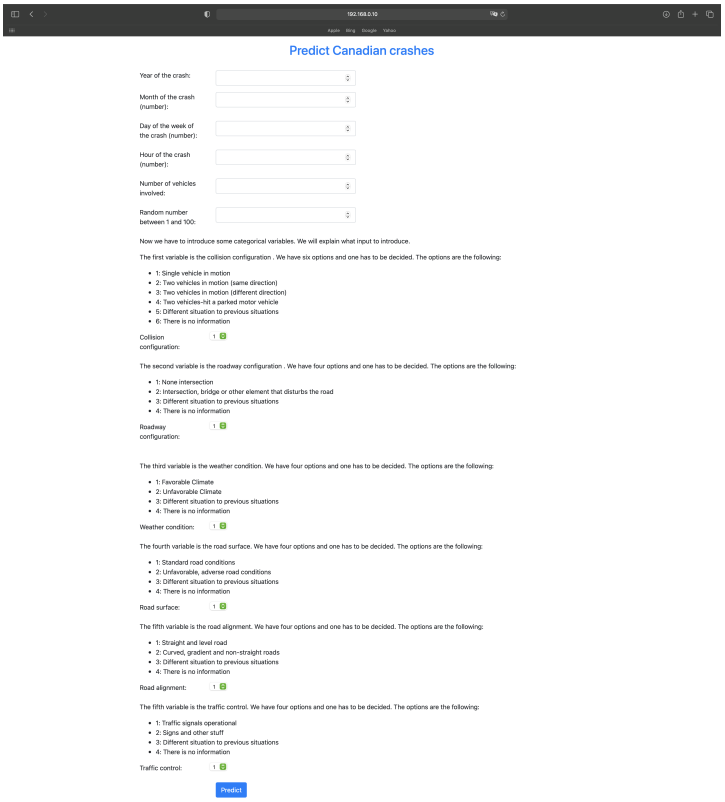

Figura 3: Html en el que el usuario introduce la información

3.2. Pruebas

A continuación realizaremos dos pruebas para comprobar que nuestra aplicación funciona correctamente.

3.2.1. Prueba 1

En esta primera prueba deberemos obtener como resultado *"habrá al menos un fallecido"* y los datos de los que haremos uso serán los mismos que los que hemos utilizado en la forma 1, en (2.2.1). Estos datos deberán ser introducidos en la figura 3.

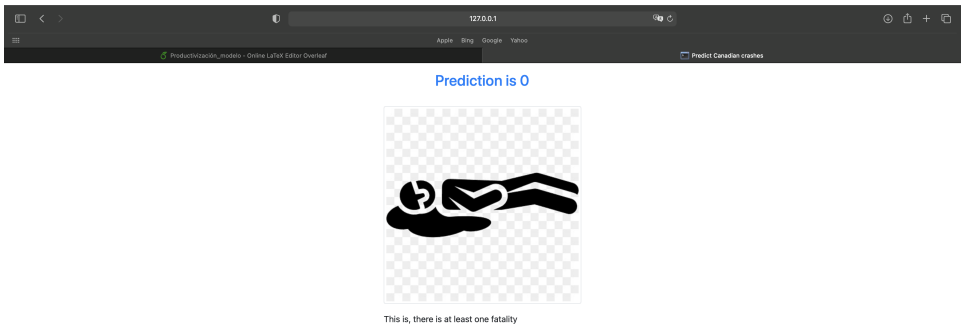

Figura 4: Resultado de la prueba 1

Podemos observar que el resultado obtenido es *"habrá al menos un fallecido"*, tal y como queríamos.

3.2.2. Prueba 2

En la segunda prueba deberemos obtener como resultado *"no habrá fallecidos"* y los datos que utilizaremos se pueden ver en la forma 1, en (2.2.2). Al igual que en la prueba 1, estos datos deben ser introducidos en la figura 3.

Podemos observar que el resultado obtenido es *"o habrá fallecidos"*.

4. Docker

Finalmente, con el fin de generar un entorno en el que poder ejecutar nuestras predicciones, haremos uso de los contenedores Docker. En ambas formas la creación de los contenedores es de la misma manera, luego, no diferenciaremos entre forma 1 y forma 2 como hemos hecho hasta ahora.

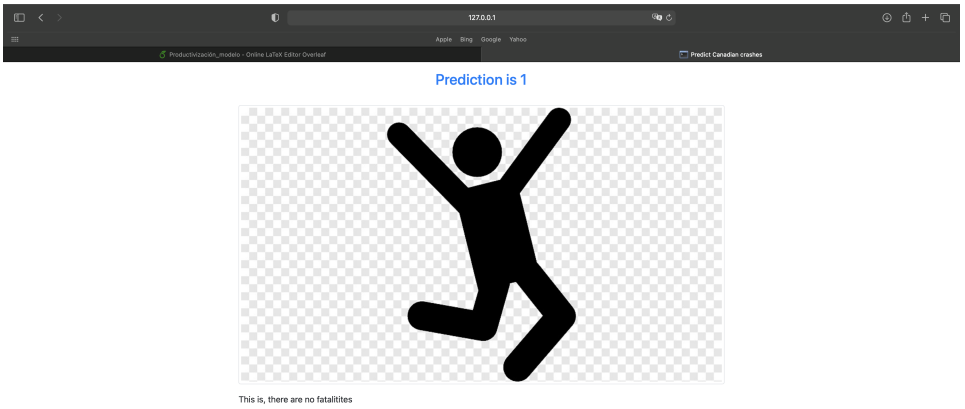

Figura 5: Resultado de la prueba 2

Los contenedores Docker sirven para resolver el problema *"funciona en mi ordenador"*. Crear un modelo que funcione y se ejecute correctamente en nuestro ordenador no tiene una complejidad extrema. Sin embargo, si queremos hacer que ese mismo modelo funcione en cualquier otra máquina en el mundo, comienza el problema. De esto último se ocupa Docker.

Luego, nuestro objetivo será crear un contenedor Docker en el que todo se ejecute y funcione de manera correcta. Para ello deberemos seguir una serie de pasos que explicaremos a continuación.

Observación. Debido a un problema en nuestros ordenadores, no hemos podido crear correctamente el contenedor docker. El problema se da cuando intentamos cargar el modelo en formato *Pickle*. Justamente el contenedor no consigue ejecutar la línea en la que cargamos el modelo y, en ese instante, el contenedor se muere. Esto es, hemos conseguido crear una imagen Docker y, después hemos ejecutado el contenedor. Pero, a la hora de ejecutar nuestra aplicación dentro del contenedor, ésta falla. Aun así, detallaremos los pasos que deben darse para crear el contenedor y que todo funcione correctamente.

4.1. Pasos a seguir

Antes de comenzar a crear nuestro contenedor Docker, crearemos un virtual environment[1] para instalar las librerias que vayamos necesitando. Para ello, primeramente, deberemos instalar la propia aplicación *virtualenv* y lo haremos introduciendo la siguiente sentencia en la terminal (en el directorio que se desee):

pip install virtualenv

Ahora nos deberemos crear el virtual environment. En nuestro caso lo llamaremos *venv_crashes_web*. La creación del viartual environment la hacemos mediante la siguiente sentencia:

[1] Un virtual environment es un entorno en el que podemos guardar los módulos que necesitemos instalar.

python3 -m venv venv_crashes_web

Nuestro objetivo con el virtual environment será instalar, en él, las librerías necesarias para que nuestra aplicación app.py funcione. Esto nos será de utilidad para la creación de nuestro contenedor Docker. Luego, para poder comenzar a instalar librerías deberemos activar el virtual environment. Lo haremos con la siguiente sentencia:

source venv_crashes_web/bin/activate

Ahora nos encontraremos dentro del virtual environment y ya podemos comenzar a instalar librerías con la sentencia *pip install librería*.

Una vez tenemos las librerías instaladas y nuestra aplicación funciona con las librerías de nuestro virtual environment, lo que debemos hacer es crear dos nuevos archivos. Estos son *requirements.txt* y *Dockerfile*. Veamos para qué sirve cada uno y que debemos introducir en ellos:

- *requirements.txt*. Este documento es un documento de texto que contiene toda la información de paquetes y sus correspondientes versiones utilizados en el virtual environment para que la aplicación Flask funcione. En nuestro caso el documento es el siguiente:

  Listing 1: Archivo requirements.txt

```
click==8.0.3
Flask==2.0.2
itsdangerous==2.0.1
Jinja2==3.0.3
joblib==1.1.0
MarkupSafe==2.0.1
numpy==1.22.0
pandas==1.3.5
python-dateutil==2.8.2
pytz==2021.3
scikit-learn==1.0.2
scipy==1.7.3
six==1.16.0
threadpoolctl==3.0.0
Werkzeug==2.0.2
```

  Este documento se crea desde la terminal. Colocándonos en la carpeta del virtual environment, la instrucción que nos crea el documento en dicho directorio es:

  pip freeze > requirements.txt

- *Dockerfile*. Este documento permite al Docker crear el contenedor de una forma determinada, es decir, el archivo Dockerfile contiene las instrucciones que le pasamos al contenedor para su creación. Veamos a continuación el archivo y expliquemos el significado de cada una de las líneas.

  Listing 2: Archivo Dockerfile

```
FROM python:3.8-slim-buster
```

```
COPY ./requirements.txt /app/requirements.txt

WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt

CMD ["python" "app.py"]
```

- *FROM python:3.8-slim-buster*. Creamos nuestra imagen considerando una imagen base ya creada. En nuestro caso, haremos uso de slim-buster ya que tiene un tamaño de 114 MB. Normalmente intentamos coger la imagen con el menor tamaño posible en la que nuestra aplicación funcione.
- *COPY requirements.txt /requirements.txt*. Copiamos el archivo de requisitos en un paso de compilación separado antes de agregar la aplicación completa a la imagen.
- *WORKDIR /app*. Creamos un nuevo directorio en la imagen que será /app.
- *COPY . /app*. Copiamos todo lo que tenemos en nuestro directorio local a la carpeta /app de nuestra imagen.
- *RUN pip install -r requirements.txt*. Con esto, le decimos al Dockerfile que instale las librerías que tenemos en *requirements.txt*.
- *CMD [”python”, .ªpp.py”]*. El comando CMD especifica la instrucción de qué se ejecutará cuando se inicie el contenedor Docker.

Una vez se tienen los archivos requirements.txt y Dockerfile, podemos comenzar a crear nuestro contenedor. La creación del contenedor la haremos mediante el siguiente comando que introduciremos en la terminal:

docker build -t flask-crashes

donde flask-crashes será el nombre de nuestra imagen. Expliquemos los comandos a continuación:

- *build*. build nos permite construir una imagen a partir de un Dockerfile.
- *-t*. -t nos permite dar nombre a la imagen y además añadirle una etiqueta. En nuestro caso no hemos dado ninguna etiqueta pero si queremos introducir una etiqueta tendríamos que haber escrito: *docker build -t flask-crashes:etiqueta*

El resultado de ejecutar el comando que acabamos de explicar es el siguiente:

Podemos comprobar que efectivamente nuestra imagen se ha creado ejecutando la siguiente sentencia en la terminal:

docker images

images nos permite mostrar una lista de las imágenes que tenemos. El resultado que obtenemos es el siguiente:

Podemos comprobar que aparece la imagen *flask-crashes* que acabamos de crear. Una vez tenemos la imagen creada, debemos ejecutarla. Lo haremos con la siguiente sentencia:

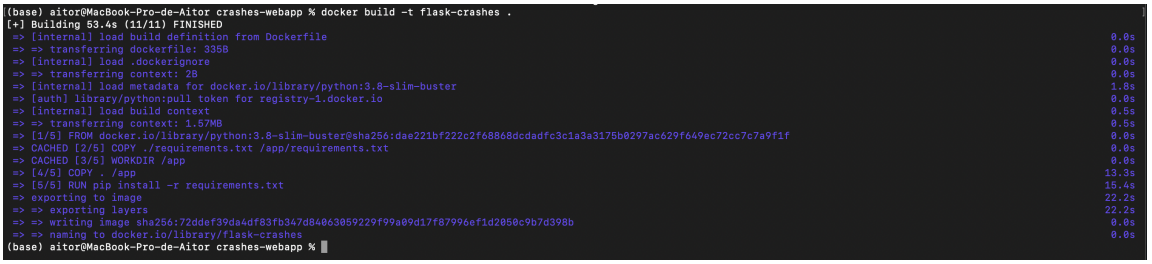

Figura 6: Imagen de la ejecución de la construcción de la imagen

Figura 7: Imagen de la ejecución de la lista de imágenes

docker run -d -p 5000:5000 flask-crashes

Expliquemos a continuación la sentencia:

- *-d*. -d nos permite ejecutar la imagen en segundo plano y mostrarnos por pantalla su ID.
- *-p 5000:5000*. Indicamos el puerto en el que queremos que se ejecute nuestra imagen.
- *flask-crashes*. Nombre de la imagen del contenedor.

El resultado obtenido al ejecutar este última sentencia se muestra a continuación:

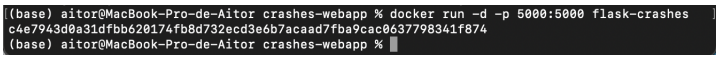

Figura 8: Imagen del resultado de la ejecución de la imagen

Vemos cómo nos aparece el ID del contenedor, luego la imagen ha sido creada correctamente. De esta forma ya hemos creado un contenedor con el que poder ejecutar todo nuestro código sin ningún problema. El siguiente paso a realizar sería adentrarnos en el contenedor y ejecutar nuestra aplicación escribiendo *python app.py* en terminal.