

MyDestiny

Víctor Cordero López
Javier Martín Martín

Proyecto final del Master in Data Science Kschool 4th edition

Introducción

A continuación se presenta una memoria del proyecto realizado de manera concisa, describiendo la metodología utilizada, así como los elementos construidos durante el desarrollo y que pueden encontrarse en el repositorio del proyecto:

<https://github.com/evolsidog/MyDestiny>

También se describirá la aplicación final que contiene la lógica necesaria para poder recomendar viajes y mostrar los puntos de interés (POI de aquí en adelante) de cada país. Para concluir se ha redactado un breve manual de usuario para poder manejarse por la aplicación.

Objetivo del proyecto

El objetivo principal del proyecto consistía en realizar una aplicación que pudiera recomendar viajes en base los gustos del usuario. También como objetivo secundario se buscaba ayudar al usuario a disfrutar del viaje recomendado mostrándole los mejores sitios para ver en ese país.

Preparación de los datos:

Los datos de los que partimos son datos de vuelos correctamente anonimizados. Al ser privados solamente comentaremos que los datos que se nos proporcionaron venían en un archivo con 66 columnas, en los que cada registro daba información acerca de cada vuelo y estaban identificados por un campo denominado “localizador”. Los datos contenían valores nulos, fechas, cadenas y valores numéricos.

Estos datos venían en un archivo comprimido, el cual contenía un csv de un peso aproximado de 3,3 GB, con un total de más de 9 millones y medios de registros.

La preparación de los datos se llevó a cabo en el notebook “*First exploratory analisis.ipynb*”.

Se hicieron varias iteraciones hasta obtener los datos que alimentarían el modelo.

- Iteración 1: Se agrupó por rloc (localizador de viaje) y por persona pensando que se podían trazar las escalas del viaje.
- Iteración 2: Se observa que no hay hora de vuelos, por lo que no se puede utilizar lo anterior. Se ignoran las conexiones y se toma cada registro como un cliente adquiriendo un producto.
- Iteración 3: Al agrupar por rloc se ve que hay registros con mismo rloc, lo cual perjudica si tomamos cada registro como un vuelo de principio a fin. Se ve que sobre un 70% tienen un solo rloc y un 9% 2 rlocs. Se ignoran el resto de registros que tengan un rloc que sea compartido más de 2 veces. Se quitan porque si tienen muchos rloc quizá solo hagan un viaje con muchas escalas, pero parecerá al tomar cada registro como un vuelo que esa persona es muy viajera, por lo que optamos por ignorar esos registros a la hora de entrenar el modelo.
- Iteración 4: Para hacer lo anterior se pensó procesarlo por trozos con pandas pero se descubre que los registros no están ordenados por rloc. Primero hay que crear un histograma con los rloc y el número de registros que tienen ese rloc recorriendo todo el conjunto de

datos 2 veces, una para crear el histograma y otra para comparar con él y quitarlos del conjunto de datos finales.

- Iteración 5: Se recupera la agrupación por rloc y persona. En lugar de coger el campo dni que no identifica de la manera que cabría esperar de manera unívoca, se escoge como campo de identificación del cliente el nombre de la persona, dado por el campo “full_name”, que no viene nunca vacío.

Finalmente el histograma nos revela que hay un total de 1042765 identificadores de viajes únicos de rloc, y nos quedamos con 5670522 registros, los cuales pasan de tener 66 columnas a 7:

- rloc: identificador de viaje
- full_name: nombre completo de la persona
- departure_date_leg: fecha de salida en formato YYYY-MM-DD
- board_point: ciudad de salida.
- board_country_code: país de salida.
- off_point: ciudad de destino
- off_country_code: ciudad de destino

Contenido del repositorio:

Sistema de recomendación:

En paralelo a la preparación de los datos se fue desarrollando el sistema de recomendación. Puede verse la versión final en el notebook “***ModeloRecomendacion.ipynb***”.

El notebook “***modelo_parte2.ipynb***” contiene un ejemplo fácil de seguir para entender mejor como funciona el modelo de recomendación.

Nuestro sistema de recomendación está basado un algoritmo de agrupación, *K-Nearest Neighbours*. Primero, con los datos preparados, agrupamos por el nombre de la persona para agrupar en un cadena todos los viajes que ha hecho, y una vez agrupados se utiliza la función dummy para crear un vector de ceros y unos con todos los posibles destinos.

Para evitar problemas con personas con viajes iguales, decidimos que nuestro conjunto de entrenamiento fueran solo con vectores distintos, por lo que agrupamos por vectores y creamos una variable peso (que es la importancia del vector), el cual se calcula mediante el número de veces que se repite el vector.

Una vez construido y entrenado el modelo con K=5 y la distancia coseno, lo guardamos en un archivo serializado utilizando la librería de *Python pickle* para ello, optimizando luego el tiempo de consulta a la hora de realizar peticiones desde la API.

Para predecir los destinos, el modelo nos da los 5 vectores más cercanos y su distancia al vector original.

Con los 5 vectores, sus distancias, sus pesos y el vector original creamos un vector nuevo aplicando la siguiente formula.

$$\text{Vector} = \text{sum}((\text{vectorPredicho}(i) - \text{vectorOriginal}) * \text{peso}(i) / \text{distancia}(i))$$

Finalmente nos quedaremos con los 5 viajes con mayor puntuación.

POI:

Como parte de la aplicación final se desarrolló en un notebook la lógica necesaria para mostrar los POI de cada país según se recomendara uno u otro utilizando las APIs de Google y la librería Folium para generar mapas interactivos que contuvieran estos POI.

Para poder utilizar la aplicación de Google es necesario estar en posesión de una clave, la cual se puede obtener desde la pagina web de *Google Places*. Para probar el proyecto no es necesario obtener clave, ya que por defecto se utiliza nuestra clave:

https://developers.google.com/places/web-service/supported_types

La url también contiene toda la información acerca de la API, como por ejemplo, tipos de puntos de interés que podemos encontrar en este caso.

Para dibujar los mapas, como se ha comentado anteriormente hemos decidido utilizar la librería *Folium*. Para la generación del mapa y obtención de los Pois, usamos tres funciones principales.

- ParseGoogleXMLResponse : que traduce un fichero xml (es el fichero descargado de la API de Google, también se puede descargar en formato json) a un diccionario formado por el nombre del pois, su tipo y calle.
- GetGoogleInfo: dada una long, lat y un radio, nos da los mejores 20 POIs dentro del círculo centrado en lat y long y radio el radio.
- find_lat_long: dada el nombre de una ciudad y el código del país, nos devuelve su latitud y longitud.

Por último tas hacer uso de las funciones anteriores, se dibuja un mapa centrado en el punto de búsqueda que hemos usado para buscar los POIs, y vamos añadiendo una marca en el mapa por cada POIs.

En el notebook “**API Google.ipynb**”, se puede probar solo la parte del código de generación del mapa con Folium y la obtenciones de los Pois. En el notebook está puesta la clave para poder hacer pruebas.

Una vez desarrollados los módulos que se integrarían en la aplicación final tocaba pensar en la persistencia de la misma. Se vio necesario alimentar una base de datos *SQLite3* con los datos de todos los países, para lo cual se desarrolló un *script* que lo insertara, el cual puede encontrarse en el notebook “**Load_Countries.ipynb**”.

MyDestiny

La aplicación final se ha construido utilizando el micro framework *flask* junto a *flask-security*. También se ha utilizado como base de datos *SQLite3* y como ORM *Flask-SqlAlchemy*, el cual es una adaptación de *SqlAlchemy* para *Flask*.

El archivo *app.py* contiene la lógica principal en el cual se crean los objetos de modelo, se inicializa la aplicación y los servicios o *blueprints* se describen al final del mismo.

Como funciones, tiene una para devolver un mapa generado con *Folium* y con los POI incluidos, otra para añadir países a un usuario, otra para borrarlo, y el método principal denominado "*suggestion*" encargado de devolver la sugerencia y cuyo flujo es el siguiente:

1. Tras comprobar que el usuario tiene países en su haber, se obtienen los códigos ISO para esos países, se construye un *array* de *numpy* que será la entrada al modelo de recomendación con todo a 0, y se ponen 1 en los países que el usuario tenga.
2. Se manda ese *array* como entrada al sistema de recomendación, y tras obtener los códigos de los países sugeridos, se genera el mapa con los POI de el país con la mejor afinidad para ese usuario.

Otros archivos

- El directorio *SQLiteConector* contiene un conector para utilizar contra la BD de SQLite3, fácilmente adaptable para otras BBDD como PostgreSQL. También tiene un script con el cual se puede generar el modelo entrenado que se utiliza en la aplicación.
- Se utilizó un archivo *.gitignore* para ignorar ciertos archivos en las subidas al repositorio.
- El notebook "***Sanic_Example.ipynb***" contiene un prototipo de Sanic, el cual implementa UVLOOP, la versión más óptima de *asyncio*, liberaría que ayudaría a resolver múltiples conexiones de manera asíncrona mediante el uso de corutinas mediante el uso de Python 3.X.

Conclusiones

Se ha construido el sistema objetivo, capaz de recomendar mediante el uso de una interfaz gráficas - una aplicación web en este caso - viajes en base a sus gustos y de ayudarle a saber qué visitar mediante el uso de los POI.

Manual de Usuario

Para que sea más sencillo probar la aplicación se ha optado por dar la opción de poder correr un contenedor de *docker* en el que un servidor *uwsgi* sirve la aplicación de *flask*.

Otra opción, por si algo fuera mal y no se pudiera construir un contenedor de *docker*, sería ir al directorio de la aplicación y ejecutar el archivo *app.py* de la aplicación *MyDestiny*:

python app.py

Para construir el contenedor de *docker* nos situamos en el directorio raíz de la aplicación en el cual se encuentra nuestro archivo *dockerfile* y ejecutamos:

docker build --no-cache --rm -t my_destiny .

Para arrancar el contenedor utilizamos:

docker run -d --name flask --restart=always --mount type=volume,src=vol_destiny,dst=/webapp my_destiny

donde:

- **-d** ejecuta el contenedor en modo *detached*. Si se quiere correr en primer plano se debe quitar este parámetro.
- **-name**: permite nombres de contenedor personalizados, donde asignamos el nombre de contenedor "flask". Si no se especifica un nombre, Docker asignará automáticamente uno.
- **-restart=always** reiniciará automáticamente el contenedor si la aplicación sale.
- **-src**: especifica el nombre del volumen que se utilizará para persistir el contenedor.
- **-dst**: indica la ruta del directorio que se quiere montar.
- **-my_destiny** especifica la imagen para el contenedor, el cual fue construida en un paso anterior.

El contenedor debería arrancar en la dirección: <http://172.17.0.2:8000>.

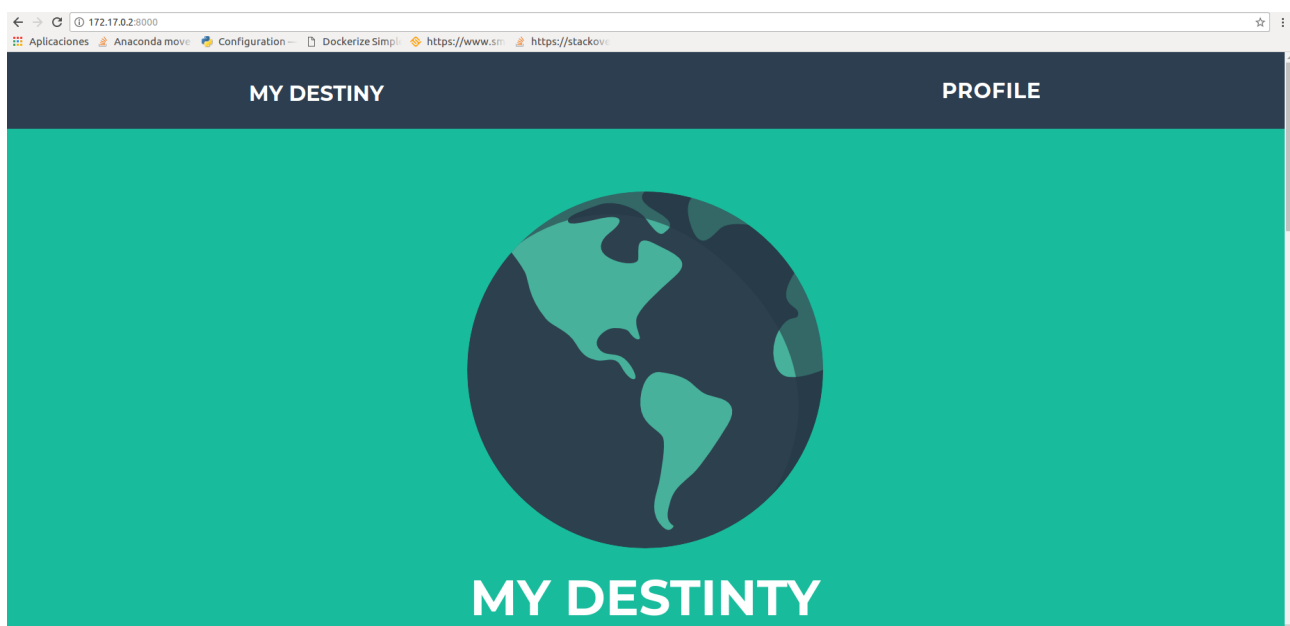
Si al ejecutar el comando:

docker ps

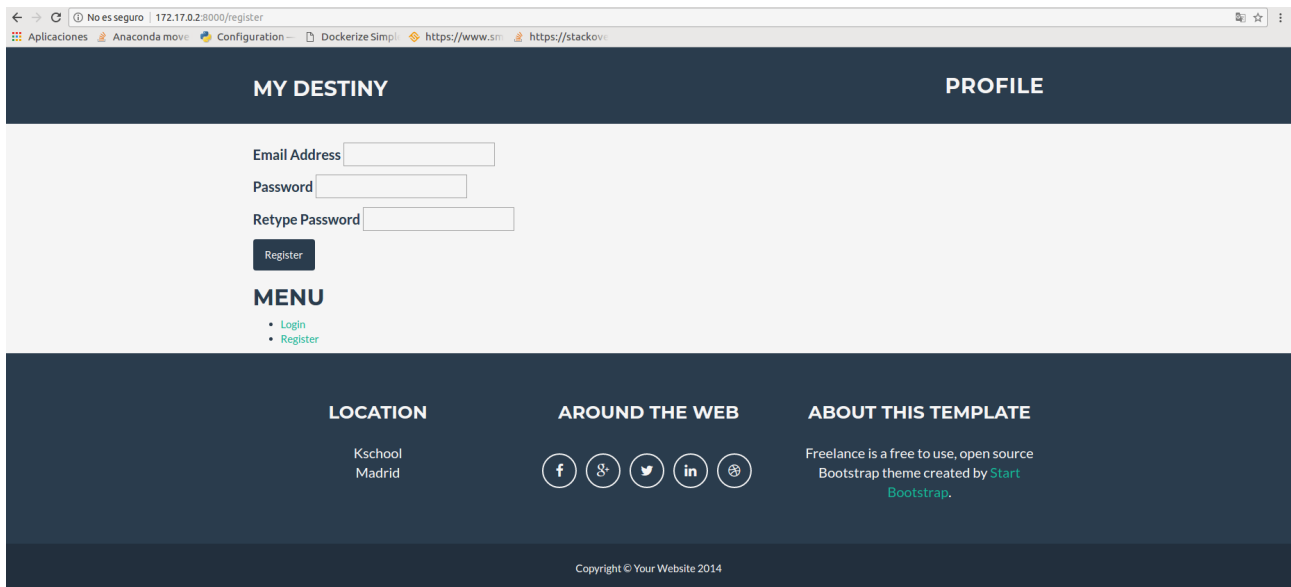
vemos que aparece el contenedor corriendo pero que no nos aparece la página principal al acceder a la url, podemos ver donde debemos acceder utilizando:

docker inspect --format '{{ .NetworkSettings.IPAddress }}' <id_container>

Una vez en la página principal de *MyDestiny* tenemos dos opciones; la primera sería loguearnos si ya estamos registrados en la aplicación. Para ello podemos ir a la dirección <http://172.17.0.2:8000/login> o pinchar en el vínculo que aparece arriba a la derecha en la página principal donde aparece la opción “**Profile**”, la cual para acceder a ella nos pedirá antes loguearnos.

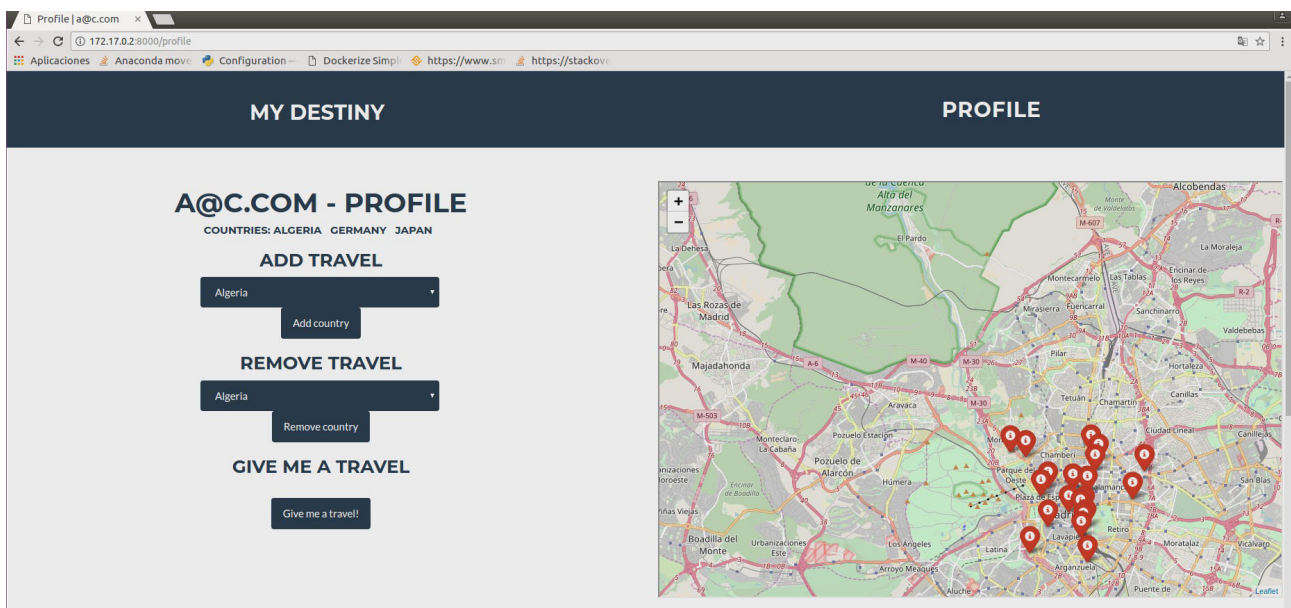


La otra opción sería registrarnos primero. Podemos acceder a la página de registro accediendo a partir de la página de *login* y seleccionando la opción debajo de la parte de credenciales, o escribiendo en el navegador <http://172.17.0.2:8000/register> .



Una vez nos registremos con nuestro correo, el cual verificará que contenga @ y que acabe con una extensión que contenga un punto, podremos acceder a la sección Profile, donde podremos buscar sugerencias de viajes.

En esta pantalla podremos seleccionar países y añadirlos o quitarlos a nuestra lista de países visitados. Por defecto nos aparecerá el mapa con los POI de España, pero en cuanto pulsemos el botón para pedir una sugerencia el mapa cambiará, mostrando el mapa del país sugerido, junto a los otros países que también nos podrían gustar.



Para cerrar la sesión de nuestro usuario podemos o bien limpiar la caché de nuestro navegador (ctrl + shift + supr) o podemos escribir en la barra de direcciones de nuestro navegador <http://172.17.0.2:8000/logout> .

Resumen herramientas utilizadas

IDE / Editor

- PyCharm
- Sublime Text
- Vim

VCS y Respositorio

- Git
- Github

Lenguajes

- Python 2.7.13
- Html
- JQuery
- Javascript
- Css

Librerías

- Flask
- Flask-Security
- Flask-SqlAlchemy
- Folium
- Jinja2
- Numpy
- Pandas
- Sklearn
- Urllib3
- Pickle

Utilidades

- Bootstrap
- VirtualEnv

Base de datos

- SQLite3
- SqliteStudio (cliente de BD SQLite3)

Despliegue

- Docker
- UWSGI

Si se requiere ver las librerías necesarias para poder ejecutar la aplicación de *flask MyDestiny* con las versiones utilizadas pueden verse en el archivo *requirements.txt* .

Bibliografía/Webgrafía

Sklearn

<http://scikit-learn.org/stable/modules/classes.html>

Pandas

<https://pandas.pydata.org/pandas-docs/stable/>

Flask tutorial

https://www.youtube.com/playlist?list=PLei96ZX_m9sWQco3fwtSMqyGL-JDQo28l

Flask

<http://flask.pocoo.org/docs/0.12/>

Flask Security

<https://pythonhosted.org/Flask-Security/>

SQLAlchemy

<http://docs.sqlalchemy.org/en/latest/>

Jinja2

<http://jinja.pocoo.org/docs/2.10/>

Flask + UWSGI + NGINX

<https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-uwsgi-and-nginx-on-ubuntu-16-04>

Sanic (UVLOOP + Flask)

<https://magic.io/blog/uvloop-blazing-fast-python-networking/>

Docker

<http://containertutorials.com/docker-compose/flask-simple-app.html>

<https://www.smartfile.com/blog/dockerizing-a-python-flask-application/>

Folium

<https://folium.readthedocs.io/en/latest/>