UNIVERSIDAD DE GRANADA

Máster Profesional en Ingeniería Informática

Cloud Computing: Servicios y Aplicaciones (2019-2020)

Despliegue de un servicio Cloud Native

Práctica 2

Mª del Mar Alguacil Camarero

Índice

1. Introducción

En esta práctica se pretende crear y desplegar un servicio Cloud Native completo de predicción de la humedad y la temperatura de la ciudad de San Francisco utilizando para ello la herramienta de orquestación Airlow. Este servicio comprende desde la adquisición del código fuente y acceso a datos hasta la ejecución de los contenedores, compuesto por una API de tipo HTTP RESTful. En este caso, se ha decidido dividir la API en dos versiones, 1 y 2, para ser desplegadas en contenedores distintos:

Versión 1 del código del servicio (ARIMA)

```
HTTP GET
  EndPoint 1 → /servicio/v1/prediccion/24horas/

HTTP GET
  EndPoint 2 → /servicio/v1/prediccion/48horas/

HTTP GET
  EndPoint 3 → /servicio/v1/prediccion/72horas/
```

Versión 2 del código del servicio (Random Forest)

```
HTTP GET
  EndPoint 1 → /servicio/v2/prediccion/24horas/

HTTP GET
  EndPoint 2 → /servicio/v2/prediccion/48horas/

HTTP GET
  EndPoint 3 → /servicio/v2/prediccion/72horas/
```

La funcionalidad de cada de los *EndPoints* anteriores es realizar la predicción de humedad y temperatura para los intervalos de 24, 48 y 72 horas. En la primera versión se utiliza el algoritmo de ARIMA aplicado a un subconjunto a partir de los datos de humedad y temperatura extraídos de [1]. De manera análoga, en la versión 2, se realiza la predicción de ambos factores empleando Random Forest.

En ambos casos, la interfaz de entrada y salida de la herramienta utilizada para predecir recibe el conjunto de datos mediante un archivo *pickle* y el intervalo de horas (24, 48 o 72) especificándoselo en la URL, y devuelve un conjunto de datos JSON con los siguientes datos:

```
[
  "hour":"05/04/2020 13:00", "temp":282.56, "hum":79.26,
  "hour":"05/04/2020 14:00", "temp":282.71, "hum":61.43,
  ...
  "hour":"07/04/2020 12:00", "temp":288.07, "hum":76.95
]
```

Estos datos devueltos por la función de predicción son enviados desde la API y los *End-Points* al usuario.

Nótese que se ha decidido incluir, además de la hora, la fecha, para proporcionar mayor claridad en los resultados y que sea más fácil la lectura de estos por parte del usuario, sobre todo, en intervalos de 48 y 72 horas en los se repiten las horas.

2. Microservicios

Como se comentó en la sección anterior, disponemos de dos versiones separadas en los siguientes microservicios:

- **APIv1:** Microservicio de predicción de la temperatura y humedad utilizando modelos pre-entrenados con ARIMA y almacenados en los ficheros *arima_temperature.p* y *arima_humidity.p*, respectivamente, contenidos en la carpeta */.models*.
- **APIv2:** Microservicio análogo al anterior que almacena el modelo entrenado con el algoritmo Random Forest en los ficheros *rf_temperature.p* y *rf_humidity.p*, respectivamente.

Ambos microservicios proporcionan un mensaje de bienvenida en la ruta raíz (como se puede observar en la figura 2.1) que nos permite el acceso a las diferentes predicciones ofrecidas, clicando sobre el intervalo deseado.

(a) APIv1

(b) APIv2

Figura 2.1: Pagina principal - Mensaje de bienvenida.

Se define la función `forecast` con el objetivo de proporcionar al usuario la predicción deseada. Para ello se captura el intervalo (`interval`) deseado de la URL como un entero mediante:

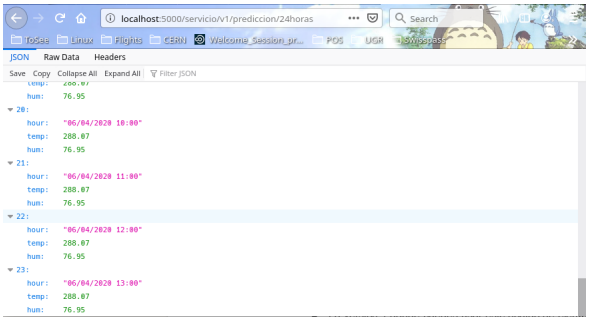
```
@app.route(/servicio/<version>/prediccion/<int:interval>horas",
           methods=['GET'])
```

Además se comprueba que dicho número sea uno de los tres intervalos permitidos, es decir, que pertenezca al conjunto {24, 48, 72}.

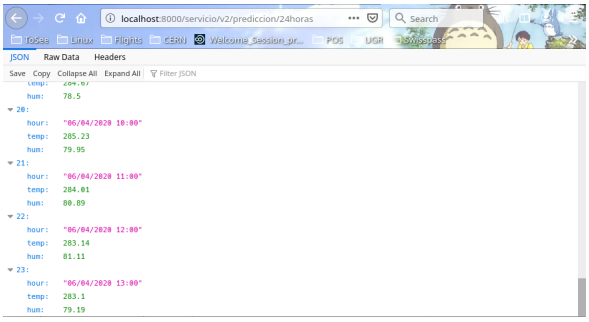
La predicción de la humedad y la temperatura se realiza mediante el modelo almacenado en los distintos ficheros que son extraídos al inicio. Sin embargo, cada algoritmo hace uso de una entrada diversa:

- **ARIMA:** basta especificar el intervalo. Las predicciones no tienen en cuenta la hora en la que nos encontramos. Si consideramos este hecho junto con que el entrenamiento se realiza con un subconjunto de 1000 datos, obtenemos unos resultados muy poco acertados donde todos las horas muestran la misma humedad y temperatura.
- **Random Forest:** necesita de la entrada de un conjunto de características numéricas. En este caso, se emplea una lista de tuplas de la forma (año, mes, día, hora) obtenida a partir de la transformación de la lista creada con las próximas `interval` horas.

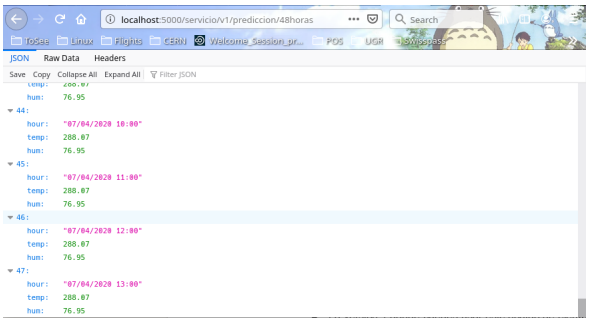
Por último, se devuelve un JSON con la fecha, la temperatura y la humedad pronosticada para las próximas horas especificadas como se muestra en la figura 2.2.



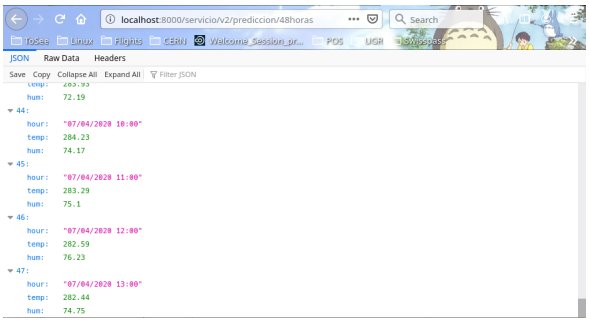
(a) Pronóstico para las próximas 24 horas. **(ARIMA)**



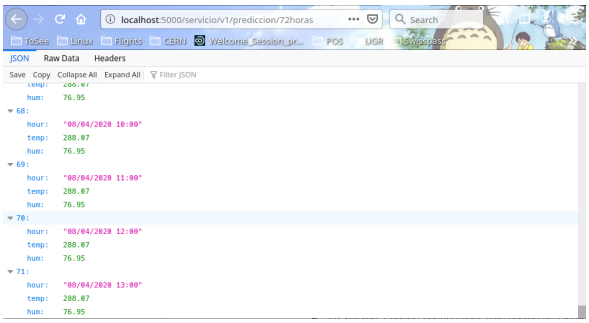
(b) Pronóstico para las próximas 24 horas. **(Random Forest)**



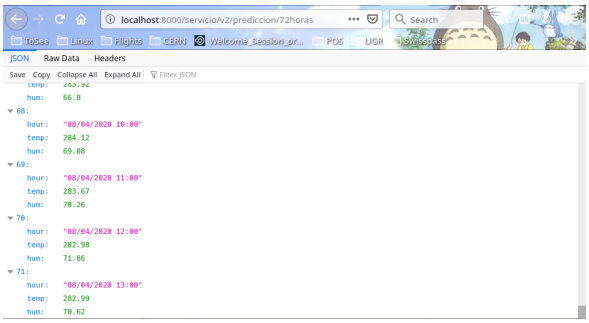
(c) Pronóstico para las próximas 48 horas. **(ARIMA)**



(d) Pronóstico para las próximas 48 horas. **(Random Forest)**



(e) Pronóstico para las próximas 72 horas. **(ARIMA)**



(f) Pronóstico para las próximas 72 horas. **(Random Forest)**

Figura 2.2: Previsiones de humedad y temperatura.

2.1. Pruebas de unidad

Como complemento a los diferentes microservicios se ha creado una batería de pruebas básicas en el fichero ***test.py*** que permite la verificación del funcionamiento correcto de cada una de las aplicaciones. Este test está compuesto por el siguiente conjunto de pruebas básicas:

- `test_index`: Verificación de la ejecución correcta de la página principal comprobando que el estado de retorno sea 200.
- `test_forecast`: Todos los *EndPoints* relacionados con los diferentes pronósticos permitidos devuelven además datos en formato JSON.
- `test_no_forecast`: No tenemos acceso al resto de intervalos, devolviendo por tanto nuestra api el estado 400.
- `test_wrong_url`: Se comprueba de una de las URLs no soportadas por el sistema devuelve el estado 404.

Este test puede ser aplicado por ambas aplicaciones, simplemente tenemos que especificar la versión que queremos testear en tiempo de ejecución.

2.2. Dockerfile

Con el objetivo de desplegar las diferentes aplicaciones, se ha definido un Dockerfile con las siguientes características:

- Se crea un contenedor con la imagen `python:3.6-slim-stretch`[1].
- Se le agrega solamente los ficheros necesarios para la instalación de las componentes necesarias y la ejecución de la aplicación deseada, por lo que debemos especificar la versión que queremos ejecutar en dicho contenedor en tiempo de construcción mediante la variable `VERSION`. Es decir, incluimos en los archivos *API$VERSION.py* y *requirements.txt* en la carpeta *workflow*.
- Se especifica que *workflow* será nuestro directorio de trabajo.
- Se crea la carpeta *.models*, la cual se utilizará para poder compartir los ficheros *pickle* que almacenan cada uno de los modelos. Además de instalar los paquetes software necesarios para la ejecución correcta de nuestra aplicación.
- Se informa a Docker que el contenedor escuchará en un puerto especificado a partir de la variable de entorno `PORT` en tiempo de ejecución.
- Por último, se despliega en el puerto `PORT` la aplicación especificada en la variable de entorno `VERSION` que toma su valor del argumento especificado en tiempo de construcción.

[1] https://hub.docker.com/_/python

3. Flujo de trabajo

A lo largo de esta sección se explicará en detalle las diferentes tareas que se han definido para la creación y despliegue de nuestro servicio Cloud Native completo de predicción de la humedad y la temperatura de la ciudad de San Francisco.

El grafo asociado al flujo de trabajo diseñado e implementado utilizando Apache Airflow se puede observar en la figura 3.1.

Figura 3.1: Grafo asociado a nuestro flujo de trabajo.

A continuación se detallan cada una de las componentes de nuestro flujo de trabajo:

- `PrepareEnviroment`: Crea la carpeta especificada en la variable de Airflow denominada `path_workflow` en caso de que no existiese. Este será nuestro directorio principal de trabajo.

  La variable `path_workflow` puede ser definida a partir del siguiente comando:

  ```
  airflow variables -s path_workflow ‘‘/tmp/workflow/’’
  ```

  y modificada vía web accediendo a la opción de `Variables` del menú `Admin`. Aunque también podemos crearla directamente aquí.

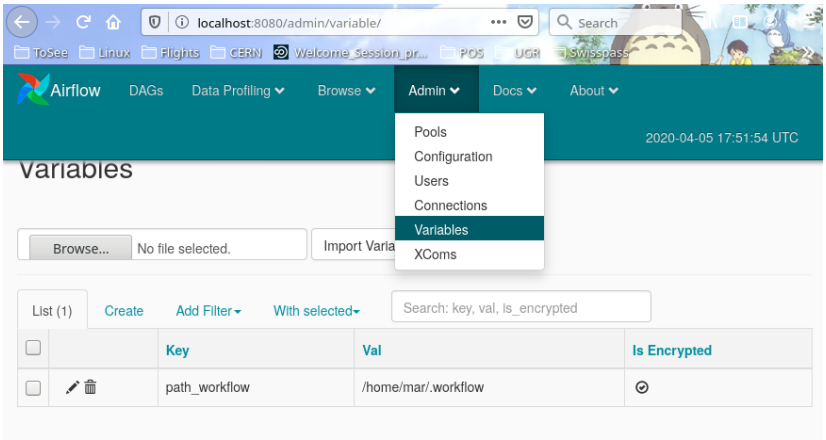


Figura 3.2: Lista de variables de Airflow.

  En este caso, se ha decidido cambiar la ubicación de la carpeta a nuestro *home* debido a problemas en la utilización de Docker fuera de esta. Se le ha añadido un punto al inicio de esta para intentar evitar que sea eliminada o modificada por el usuario mientras se ejecuta el proceso.

- `GetDataHumidity`/`GetDataTemperature`: Descarga y descomprime los datos relacionadas con la humedad y la temperatura en nuestro directorio de trabajo.

- `ProcessData`: Los ficheros de datos capturados en el proceso anterior son unificados mediante la columna `datetime` extrayendo sólo la información referente a San Francisco, tanto para la humedad como para la temperatura. Posteriormente se realiza una limpieza eliminando las filas que contienen datos perdidos y se procede a almacenar en MongoDB el nuevo conjunto de datos en la colección `hum_temp` del conjunto de datos `dataset` con el siguiente formato:

  `DATE;TEMP;HUM`

  Con este propósito se han creado las siguientes dos funciones:

  - `selectCSVcolumns`: Extrae las columnas especificadas en `column_name` del archivo CSV `csvfile`.
  - `mergeDataSets`: Realiza la fusión de datos antes explicada de los ficheros `hum_file` y `temp_file` ayudándose de la función anterior, y almacena el nuevo conjunto en la base de datos.

- `TrainARIMA`: Extrae el conjunto de datos de la base de datos, crea los modelos entrenados con ARIMA, tanto para temperatura como para humedad, y los almacena en el fichero de *arima_temperature.p* y *arima_humidity.p*, respectivamente. Para ello se define la función de Python `trainARIMA`.

  Dichos archivos son almacenados en la carpeta *.models* ubicada en nuestro *home*, que se crea automáticamente si no existe.

- `TrainRF`: Análogamente al caso anterior se crean los modelos de Random Forest a partir de la función `trainRandomForest`. Sin embargo, en este caso, la función de predicción `RandomForestRegressor` requiere de un conjunto de características numéricas que le debemos proporcionar junto con las etiquetas (que en este caso son los datos de humedad o temperatura, dependiendo del modelo que queramos obtener). Por lo tanto, debemos transformar la fecha asociada a cada valor a una tupla de la forma *(año, mes, día, hora)* ya que los minutos y los segundos no nos interesa. Creamos una lista con dichas tuplas obtenidas teniendo en cuenta que la fecha se interpreta como una cadena de texto con la siguiente forma:

  "%Y-%m-%d %H:%M:%S"

- `CloneRepo`: Clona el código fuente asociado con nuestros microservicios en la carpeta *services* dentro de nuestro directorio de trabajo.

- `RunUnitTestsV1`/`RunUnitTestsV2`: Ejecuta las pruebas de unidad de ambas versiones para verificar que todo está correcto y se puede proceder a su despliegue.

- `DeployAPIv1`/`DeployAPIv2`: Una vez comprobado que todo funciona correctamente, se construye y despliega cada uno de los microservicios en contenedores separados. La versión 1 tendrá asociada el puerto 5000 mientras que la versión 2 el 8000. Además se liga ambas carpetas *.models* utilizando la opción `-v` que nos proporciona Docker.

- `CleanUp`: Por último, se procede a la limpieza de los archivos intermedios creados y que ya no son de más utilidad. Se elimina la carpeta especificada en la variable de Airflow *path_workflow* empleando para ello la función de python `rmtree` que nos proporciona el módulo `shutil` de Python.

4. Conclusiones

A lo largo de este trabajo hemos visto como se ha realizado la creación y despliegue de un servicio Cloud Native completo, intentando aprovechar las ventajas que nos proporcionaba Airflow para la paralelización de tareas y utilización de diferentes lenguajes de programación para la definición de las mismas. Se comprobó además que Airflow proporciona una forma fácil y dinámica de realizar todo esto. Sin embargo, el tiempo que emplea en completar el grafo de forma secuencial es mayor de lo esperado, debido probablemente a la asignación de trabajo que debe realizar internamente.

Otro de los puntos en contra, a mi parecer, es que se debe estar constantemente recargando la página para conocer el estado actual de la ejecución, además de que cuando detecta un problema se queda bloqueado el *scheduler* por unos minutos antes de dar error, error que se muestra accediendo a la subcarpeta de *logs* ubicada dentro la carpeta de *airflow*.

Probablemente la ejecución en paralelo será más prometedora que la obtenida con la secuencial. Sin embargo, se ha detectado que la secuencia de tareas mostrada en la figura 4.1 no pueden ser especificadas directamente de la siguiente manera:

```
CloneRepo >> [RunUnitTestsV1 >> DeployAPIv1, RunUnitTestsV2 >> DeployAPIv2]
```

sino que debemos utilizar los operadores `set_downstream` (empleados en este proyecto, como se puede ver aquí) o `set_upstream`.

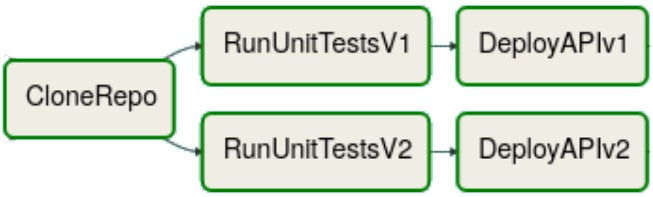


Figura 4.1: Secuencia de tareas.

Además se ha comprobado que no puede concatenar dos listas como se pretendía realizar con las lista de tareas de [`TrainARIMA`, `TrainRF`, `CloneRepo`] y [[`RunUnitTestsV1 >> DeployAPIv1`, `RunUnitTestsV2 >> DeployAPIv2`]].

A pesar de estas pequeñas deficiencias que parece tener, se considera que puede ser una herramienta de mucha utilidad en proyectos de mayor envergadura.

En lo que respecta al resto de herramientas utilizadas para este proyecto se han ido aprendiendo conforme se han ido necesitando. En general, no se han tenido grandes problemas que *stackoverflow* no pudiese solucionar, salvo en el caso de Docker que no me permitía utilizar volúmenes debido a que la carpeta base estaba fuera del *home*. Al no dar ningún error en tiempo de ejecución del comando, fue difícil descubrir la razón del problema de conexión entre ambas carpetas.

Referencias

[1] Repositorio con el material proporcionado para las prácticas de la asignatura Cloud Computing: Servicios y Aplicaciones.
https://github.com/manuparra/MaterialCC2020

[2] *(Airflow) Concepts.*
https://airflow.apache.org/docs/stable/concepts.html?highlight=branch

[3] *pickle — Python object serialization.*
https://docs.python.org/3/library/pickle.html

[4] *Tutorial: Introduction to working with MongoDB and PyMongo..*
https://api.mongodb.com/python/current/tutorial.html

[5] *Insert Pandas Dataframe into Mongodb: In 4 Steps Only.*
https://www.datasciencelearner.com/insert-pandas-dataframe-into-mongodb/

[6] *docker build.*
https://docs.docker.com/engine/reference/commandline/build/

[7] *Docker run reference.*
https://docs.docker.com/engine/reference/run/

[8] *Function* `shutil.rmtree`.
https://docs.python.org/3/library/shutil.html#shutil.rmtree