UNIVERSIDAD DE GRANADA

Cloud Computing

Despliegue de un servicio Cloud Native

Práctica 2

Víctor Vázquez Rodríguez
victorvazrod@correo.ugr.es
76664636R

Máster universitario en Ingeniería Informática

Curso 2019/20

Índice

1 Introducción

En esta práctica se pedía construir un servicio de predicción meteorológica siguiendo el modelo conocido como ***Cloud Native***. Para ello, se hace uso de la herramienta **Apache Airflow**[1], la cuál permite definir flujos de trabajo con tareas atómicas que se pueden ejecutar de forma distribuida, aprovechando las características de la nube.

En concreto, el servicio debe permitir obtener las predicciones de temperatura y humedad para las próximas 24, 48 y 72 horas. Hay que implementar dos versiones del mismo, usando en la primera un modelo de análisis de series temporales llamado ARIMA. Para la segunda versión del servicio, he decidido usar un modelo autorregresivo. Los dos servicios se ejecutarán usando contenedores.

Ambos modelos se entrenan con los conjuntos de datos proporcionados para la práctica, los cuáles se deben procesar previamente en el flujo de trabajo desarrollado.

A lo largo de este documento se presentarán las distintas tareas que se debían realizar para la práctica, explicando la solución elegida e incluyendo enlaces a la implementación de ésta, que se encuentra en GitHub[2].

[1] Web de Apache Airflow
[2] Repositorio del proyecto

2 Conjunto de datos

Como ya hemos indicado, los modelos que usan los servicios que se han implementado para esta práctica se deben entrenar con un unos conjuntos de datos proporcionados. Estos datos, que están en formato CSV, deben descargarse, descomprimirse, limpiarse y, por último, introducirse en una base de datos.

2.1 Preparación del directorio de trabajo

Antes de realizar ninguna tarea, debemos crear el directorio en el que se va a desarrollar toda la actividad del flujo de trabajo. En mi caso, trabajaré con el directorio `/tmp/forecast`, dentro del cuál tendremos `data/` para los ficheros de datos, `models/` para los modelos y `code/` para el código fuente de los servicios.

Creamos todas estas carpetas, excepto la última (se crea al clonar el repositorio), usando un *BashOperator*.

- Creación de la estructura de carpetas necesaria

2.2 Descarga y descompresión

Al tratarse de dos ficheros, su descarga y descompresión se puede realizar en paralelo. Para realizar estas tareas, usamos *BashOperator* junto con las herramientas *curl* (para la descarga) y *unzip* (para la descompresión).

- Descarga de ficheros
- Descompresión de ficheros

2.3 Procesamiento de los datos

Los ficheros proporcionados contienen datos de múltiples ciudades, teniendo que quedarnos nosotros con los referentes a San Francisco. Además, hay que juntar los datos extraídos de ambos ficheros en uno solo usando la fecha de cada medición como clave para la unión y guardarlos en un nuevo fichero CSV.

Se ha implementado una función en Python que realiza este procesamiento a través de un *PythonOperator* en el flujo de trabajo.

- Operador del flujo de trabajo
- Función que procesa los datos

2.4 Inserción en base de datos

Por último, los datos ya procesados se insertan en una base de datos para facilitar el posterior entrenamiento de los modelos con los mismos. La base de datos elegida es una PostgreSQL[3], que lanzaremos en un contenedor Docker

[3] Web de PostgreSQL

usando su imagen oficial. Los datos se insertan usando una función Python a la que se le indica la ruta del fichero CSV donde están los datos ya procesados.

- Lanzamiento de la base de datos y ejecución del procesamiento
- Lectura del CSV e inserción en la base de datos

3 Entrenamiento de los modelos

Una vez procesados los datos y almacenados en la base de datos, se pueden usar para entrenar los modelos, tanto ARIMA como el autorregresivo. En el planteamiento inicial de la práctica, se proponía realizar este entrenamiento dentro de los propios modelos, no obstante, este es un proceso que consume mucho tiempo, por lo que he pensado que sería más eficiente entrenar los modelos en tareas separadas del flujo de trabajo y almacenar los modelos ya entrenados en ficheros, que luego se pasarían a los servicios al lanzarlos.

3.1 ARIMA

Para entrenar el modelo ARIMA, utilizamos el código proporcionado, el cuál hace uso de la librería *pmdarima*. Debemos entrenar dos modelos, uno para la temperatura y otro para la humedad. Una vez entrenados, usamos los *pickles* de Python para guardarlos en la carpeta `/tmp/forecast/models/arima`.

- Operadores del flujo de trabajo
- Entrenamiento de los modelos ARIMA

3.2 Modelo autorregresivo

El modelo autorregresivo se entrena con la librería *statsmodels*, que contiene gran cantidad de modelos estadísticos. El procedimiento es análogo al entrenamiento de los modelos ARIMA, necesitando también entrenar dos modelos y almacenándolos usando *pickle*, esta vez en la carpeta `/tmp/forecast/models/autoreg`.

- Operadores del flujo de trabajo
- Entrenamiento de los modelos autorregresivos

4 Construcción y despliegue de los servicios

Los ficheros con los modelos entrenados son usados por los servicios de predicción en tiempo de ejecución. Estos servicios han sido implementados en Python con la librería ***hug***[4] y su despliegue se realiza mediante contenedores Docker.

4.1 Funcionamiento

Cada servicio consta de dos partes: el punto o ruta de acceso y el modelo predictivo. Para el punto de acceso o *endpoint* se define un controlador con *hug*, como ya hemos indicado. Este controlador obtiene el número de horas al que se quiere realizar la predicción y, con él, llama al modelo para que realice la predicción.

Al iniciar el servicio, los modelos se cargan a partir de los ficheros dados. Cuando realiza la predicción, los resultados devueltos por los modelos de temperatura y humedad se combinan en una única lista que se devuelve en formato JSON.

Como se trata de dos servicios separados y que se ejecutan en contenedores independientes, se tiene que implementar un *gateway* que sirva como punto de acceso unificado para ambos. Sencillamente, es un *proxy* inverso que redirige las peticiones recibidas a cada servicio dependiendo de la ruta indicada. Este *gateway* se ha implementado en Go.

- API de la versión 1 del servicio
- API de la versión 2 del servicio
- Modelo de la versión 1 del servicio
- Modelo de la versión 2 del servicio
- *Gateway* del sistema

4.2 Descarga del código fuente

Todo este código se encuentra alojado en GitHub, por lo que para incorporarlo en el flujo de trabajo tan solo tenemos que clonar el repositorio mediante un *BashOperator*. El repositorio se clona al directorio `/tmp/forecast/code`.

- Clonación del repositorio de GitHub

4.3 Ejecución de pruebas

Se han implementado con *unittest* algunas pruebas unitarias sencillas para la función de los modelos que se encarga de realizar la predicción y unir los resultados de temperatura y humedad. Antes de desplegar los servicios, se ejecutan estas pruebas para comprobar que su implementación es correcta.

[4] Web de *hug*

- Pruebas del modelo ARIMA
- Pruebas del modelo autorregresivo
- Ejecución de las pruebas

4.4 Construcción de imágenes

Tanto el *gateway* como las dos versiones de los servicios tienen su propio fichero `Dockerfile` que define la construcción de las imágenes de los contenedores que se lanzarán.

Todas estas imágenes exponen el puerto 8080, adonde deben llegar las peticiones. Además, las imágenes de las dos versiones de la API definen un punto para montar un volumen que es el que debe contener los ficheros de los modelos.

En el caso de la imagen del *gateway*, aprovechamos que Go es un lenguaje compilado para aplicar una construcción multifase, lo que nos permite tener una imagen final de tamaño mínimo.

Las imágenes se construyen usando `docker build` con *BashOperator*.

- Dockerfile de la versión 1 de la API
- Dockerfile de la versión 2 de la API
- Dockerfile del *gateway*

4.5 Creación de red Docker

Cada una de las partes del sistema (servicios y *gateway*) se ejecuta en un contenedor independiente. Es por ello que necesitamos crear una red Docker que permita al contenedor del *gateway* comunicarse con los de los servicios.

- Creación de la red Docker

4.6 Lanzamiento de los contenedores

Con las imágenes construidas y la red Docker creada, ya podemos lanzar los contenedores del sistema. Para ello, usamos `docker run`, teniendo en cuenta que todos los contenedores deben conectarse a la red (Con `--network`) y que el contenedor del *gateway* debe enlazar su puerto 8080 con un puerto local (con `-p`).

El orden en el que se despliegan estos contenedores también es importante, ya que el *gateway* depende de los servicios, por lo que éstos deben desplegarse primero.

- Ejecución de los contenedores

5 Flujo de trabajo completo

Todas las tareas que hemos ido indicado durante este documento se deben incorporar al grafo que representa el flujo de trabajo que se ejecuta en Airflow. El grafo final que he implementado se puede ver en la figura 1.

- Definición de las dependencias entre tareas

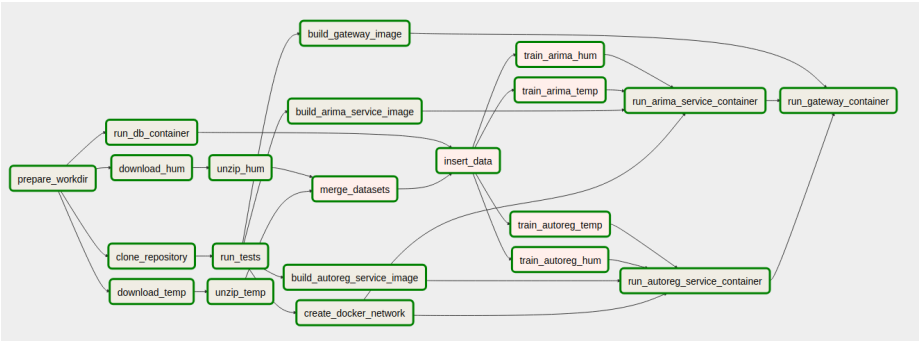

Figure 1: Grafo del flujo de trabajo.