Un nuevo mito para nuestra causa.

Introducción

En este proyecto partiremos del sistema inicial entregado por "Seedtag" y lo ampliaremos hasta conseguir que el modelo nos devuelva un listado de personajes que serán utilizados en nuestras futuras campañas publicitarias.

El sistema inicial entregado por Seedtag estaba formado por los siguientes contendores:

- Zookeeper
- Kafka
- Data-streaming-service
- Listener-service
- Clasification-service

Este sistema ha sido ampliado con otros dos contendores:

- Character-service
- Postgres-db

Adicionalmente, se ha definido la carpeta "Database Access" que contiene una seríe de Notebooks encargados de acceder a la base de datos para recuperar la información y presentarla en un formato de tabla.

La estructura del resto del documento será la siguiente, cada uno de los contenedores que forman nuestro sistema será explicado en secciones independientes, haciendo énfasis en que elementos han sido actualizados, qué elementos son nuevos y qué posibles requisitos adicionales son necesarios para la ejecución correcta del sistema. Como último apartado de estas secciones los posibles trabajos futuros a realizar y posibles mejoras.

Dispondremos de una sección adicional en la cual se explicará como funcionan los notebooks y la salida esperada en ellos, junto con una sección final, que a modo de resumen, unificará las necesarias tareas de configuración necesarias para la correcta ejecución del sistema.

Zookeeper & Kafka

Estos dos contenedores son el nucleo básico del projecto, y son los encargados de

gestionar el sistema de mensajería utilizado. Estamos utilizando las últimas versiones publicadas por confluence, y las configuraciones sugeridas por "Seedtag" han sido mantenidas.

Data-streaming-service

Este contenedor es el productor de nuestro sistema. Entre sus atribuciones está la generación del "Topic", con su configuración específica, el cual será utilizado para la comunicación de los mensajes. Además, se encargará de la generación de cada uno de los mensajes que serán trasmitidos.

Dockerfile

Este archivo es el encargado de cargar el contenedor en nuestro sistema, y de realizar las configuraciones necesarias. En nuestro caso no ha sido modificado y es el mismo entregado por "Seedtag".

En él se decide que la máquina virtual debe de ejecutar el archivo "launch.sh".

launch.sh

Como en el caso del archivo Dockerfile explicado anteriormente, no ha sido modificado. Su función es la de ejecutar los dos archivos Python que se encargan de inicializar el Topic y de generar los mensajes.

kafka_topic.py

Se encarga de conectarse con Kafka y definir nuestro nuevo Topic llamado "events". Además, define el número de particiones que serán utilizadas junto con el factor de replicación.

Este es otro archivo que no ha sido modificado.

kafka_events.py

Este es el archivo encargado en generar los mensajes y añadirlos al Topic. Los mensajes se generan a partir de los textos contenidos en la carpeta "texts". El archivo lee, de forma ininterrumpida, los archivos contenidos en enta carpeta y genera un mensaje por cada uno de ellos. La lectura ininterrumpida de la carpeta provoca que múltiples mensajes sean definidos a partir del mismo texto. Esto será tomado en cuenta en futuras decisiones tomada para la correcta ejecución del sistema.

En este caso, si ha sido modificado ligeramente la función delivery_report. No ha sido

cambiada en absoluto su funcionalidad solo ha sido aumentada la información publicada en el registro en previsión de que el sistema utilizara múltiples topis o multiples particiones.

Posibles modifcaciones a realizar.

Entendemos que la instrucción "p.flush()" sirve para retener el proceso a la espera de que el mensaje sea correctamente enviado. Por lo tanto, consideramos que debería de estar contenida dentro del bucle "while" y no estar con posterioridad al mismo.

Listener-service

En este contenedor tenemos definido el consumidor que recibirá los mensajes contenidos en el Topic "events". Inicialmente, este contenedor solo recuperaba los mensajes y mostraba el resultado en la terminal.

En nuestro caso hemos modificado ampliamente el funcionamiento del mismo. Ahora se comunica con los contenedores classification-service y character-service para obtener información adicional que posteriormente guardará en la base de datos.

docker-compose.yml

La entrada donde se define el listener-service ha sido modificada introduciendo dependencias tanto sobre el classification-service y el character-service, ya que depende de estos dos contenedores para poder obtener toda la información necesaria. Y también añadimos una dependencia sobre el contenedor postgres-db que contendrá una base de datos PostgreSQL.

Dockerfile

Entre los nuevos requisitos para la correcta ejecución de este contenedor, tenemos definidas la necesidad de comunicarse con los contenedores:

- classification-service
- character-service
- postgres-db

Por lo tanto, ha sido necesaria la instalación de las siguientes librerías:

- requests: Para realizar un acceso via HTTP
- psycopg2-binary: Para acceder a PostgreSQL

Una libreria menos crítica para la ejecución del contenedor, pero que también tenemos que importar es "JSON". La cual es necesaria para interpretar el contenido del mensaje y

transformarlo desde un texto a un diccionario y darle la necesaria estructura.

kafka_consumer.py

La mayor parte de las modificaciones realizadas en el proyecto se han realizado en este archivo, ya que como hemos comentado anteriormente, se ha ampliado su lógica.

Clasification-service

Debemos de comunicarnos con el contenedor clasification-service para conocer a que "saga" pertenece el texto ingresado en el mensaje.

character-service

Nos comunicamos con este contenedor para obtener los personajes contenidos en el texto. Debemos de recalcar que esta información es básica para nuestro projecto.

Salida por consola

Además, ha sido modificada la salida que obtenemos por la consola durante la ejecución del contenedor.

Lo primero que hemos modificado ha sido la presentación de la información de cada mensaje, separándolos entre ellos para que sea más sencillo su localización y lectura.

Como el consumidor puede estar esperando la llegada de nuevos mensajes a través del Topic, hemos añadido un mensaje que indica que el consumidor está a la espera de recibir los mensajes.

Y la información relativa a los mensajes, junto con la información descargada de los servicios, ha sido reformateada para una lectura más sencilla. De la misma forma, se ha reducido el texto presentado y sacaremos por pantall solo los 100 primeros caracteres, y no la totalidad del mismo. Ya que consideramos que con el identificador del mensaje podremos acceder a la totalidad del texto contenido en la carpeta "data-streaming-service/texts"

Configuración

Aparte del conjunto de librerías adicionales, las cuales se instalaran de forma automática en el contenedor, necesitaremos realizar una acción manual, que será la modificación de la IP local del equipo en el cual se están ejecutando los contenedores. La nueva IP será necesaria introducirla en la línea 22 del texto. ´´´ ip = '192.168.178.35´ ´´´

Seguridad

Realmente al ser un prototipo no hemos trabajado en la seguridad del sistema, lo cual se puede considerar un error. Lógicamente, a la hora de poner este proyecto en producción el password y usuario que definen el acceso a la base de datos debería de ser tratado de otra forma. Esta información es visible desde la línea 28 a la 32.

Ejecución

Aunque se ha modificado la lógica del contenedor, no ha sido modificado como debe de ser levantado y arrancado el mismo.

Para levantarlo, una vez posicionado en la carpeta que contenga al "docker-compose.yml", debemos de ejecutar

- docker-compose exec listener-service bash

Y para arrancar el servicio

- python3 kafka_consumer.py

Classification-service

Este contenedor contiene un modelo llamado "classification_pipeline.joblib" que se encarga de predecir a qué saga pertenece el texto que le damos. Inicialmente, el servicio estaba preparado para ser accedido a través de una consulta HTTP, y nosotros no hemos modificado esto.

De esta forma, nuestro contenedor listener-service podrá acceder a este servicio y obtener la saga.

Modificaciones

En su momento tuvimos problemas para que el servicio se levantara correctamente, y concluimos que sería necesario modificar las librerías importadas. Inicialmente, el servicio intentaba instalar la libreria "sklearn", pero esto generaba un error. Ya que se debería de importar la libreria "scikit-learn" aunque a la hora de realizar la importación de la misma, la llamaremos como "sklearn".

Para simplificar el acceso al listado de las librerías necesarias para la instalación se ha generado un archivo "requirements.txt" que contiene el listado de las librerías. Además, era necesario la importación de la version 1.20 de la libreria numpy, tal y como queda reflejado en el archivo.

Lógicamente, en el archivo "Dockerfile" utilizamos este archivo para la instalación de los paquetes adicionales.

Character-service

Este contenedor es una réplica del contenedor "Classification-service". En este caso el contenedor va a devolvernos el listado de personajes contenidos en el texto.

Para realizar esta búsqueda vamos a utilizar la libreria "nltk". Con ella vamos a realizar una tokenización del texto y posteriormente compararla con el listado de personajes contenido en el archivo "characters.txt".

Este contenedor es una copia del anterior, con la única diferencia de que ahora está escuchando en el puerto 5010, y que la url de la aplicación es "/characters". Realmente este nuevo contenedor no es imprescindible, hubiera sido posible levantar ambos servicios en el mismo contenedor.

requierements.txt

En este caso, además de las librerías necesarias para levantar la aplicación. Solo necesitamos la libreria "nltk" para la tokenización del texto.

Problemas

Al utilizar la tokenización, extraemos del texto cada una de las palabras que lo componen, desgraciadamente, a la hora de comparar estos tokens con los nombres de los personajes, nunca localizaremos los personajes que sean referenciados por nombre y apellido o que tengan un nombre compuesto.

Siguientes pasos

Sería necesario una restructuración del archivo "characters.txt", ya que hay elementos repetidos, e incluso pudiera ser interesante dividirlo en tres, de forma que cada archivo contenga solo los personajes de una saga.

Además, hay que buscar algún método para trabajar sobre los personajes definidos con dos palabras, como por ejemplo transformando a "San Gamgee" en "San-Gamgee" para que la tokenización nos devuelva un único elemento. Además de unificar todas las posibles referencias a cada personaje para que se contabilicen como una sola.

postgres-db

Hemos decidido que la mejor forma de almacenar los datos para un futuro trabajo sobre ellos es utilizar una base de datos. En este caso en particular nos hemos decidido por una base de datos PostgreSQL aunque no sería la decisión más adecuada a la hora de poner el

sistema en producción. Para poner en producción un sistema de buckets S3 considero que sería la mejor opción, ya que nos permitiría almacenar los datos en un formato JSON más apropiado para almacenar el diccionario que devuelve el sistema.

Configuración.

Vamos a diferenciar entre la configuración del contenedor que contiene la base de datos, y la base de datos en sí.

docker-compose.yml

Aquí debemos de definir las variables básicas para asegurar que la base de datos funciona correctamente, no es necesario configurar nada relativo al contenedor, ya que utilizaremos el contenedor proporcionado por PostgreSQL directamente.

Entre las variables a configurar serían:

- POSTGRES_DB
- POSTGRES_USER
- POSTGRES_PASSWORD
- Puerto de escucha

new_tables.sql

Una vez arrancada la base de datos, el siguiente paso es asegurarnos que siempre tendremos las tablas necesarias para la correcta ejecución del sistema.

En nuestro caso tendremos dos tablas, characters y documents.

documents

En esta tabla vamos a almacenar la información contenida en el mensaje generador por el productor del sistema, en el cual habremos añadido la predicción de la saga a la que pertenece. Para asegurarnos que no hay problemas de colisión entre las posibles diferencias instancias del texto, hemos definido un identificador adicional message_id que se compone con la agregación del identificador del archivo junto con el timestamp.

La definición de la tabla sería

campo	Description
id	Key data
message_id	A unique identifier that define the instance of the text
document name	The name of the file that contains the text
time	Message timestamp
readers	Number of readers of the document.
saga	Saga to which the document belongs.

characters

En esta segunda tabla vamos a almacenar los personajes que aparecen en los textos. La idea de utilizar dos posibles tablas era para reducir lo más posible la cantidad de datos repetidos, a la vez de que se pueda acceder de una forma sencilla al listado de los personajes.

La definición de la tabla sería

campo	Description
id	Key data
message_id	A unique identifier that define the instance of the text
document name	The name of the file that contains the text
charac	The characters that appear in this text

Uso

Realmente este contenedor está pensada para poder acceder a él desde el exterior para poder obtener los datos en cualquier herramienta, lo cual simplifica ampliamente su utilización y la extracción de los datos.

Persistencia de los datos

Durante la configuración de la base de datos no consideramos necesario unir la carpeta que contendrá los datos en el contenedor con una carpeta de nuestro sistema para asegurar la persistencia de los datos. La razón de ello es que este projecto es una PoC y los datos obtenidos aquí no son válidos.

Además, el sistema genera la suficiente cantidad de datos en solo cinco minutos de ejecución que ya nos permitiría comprobar la utilidad de los notebooks.

Problemas adicionales

Desgraciadamente, las consultas sobre PostgreSQL nos devuelven los datos línea a línea lo que provoca que el tiempo de ejecución no sea el más eficiente.

Database Access

Esta carpeta no contiene ningún contenedor, realmente es una carpeta que contiene tres notebooks. Dos de ellos acceden a cada una de las tablas que tenemos, para obtener información por separado. Y el tercero accede al INNER JOIN de ambas tablas.

Adicionalmente, tenemos el fichero library.py que contiene una función que transforma el resultado descargado desde PostgreSQL y lo transforma en un dataframe para simplificar la visualización de los datos

Todos los notebooks necesitan importar la librería psycopg2 que nos asegura la comunicación con PostgreSQL

Join DataSet

En este caso es el único notebook donde tenemos consultas más complejas ya qué uniremos los datos de ambas tablas. Para realizar esta unión simplemente debemos de utilizar hacer un math con el nombre del documento. Ya que nos hemos asegurado con anterioridad que en la tabla de characters no contenga documentos duplicados.

Configuración

Como con anterioridad, será necesario modificar el parámetro IP en las conexiones para asegurar que los notebooks acceden al contenedor sin problemas. Como con anterioridad debemos de asignar el número IP de la máquina en la que estemos ejecutando el contenedor.

Futuros trabajos

Aunque los Notebooks es una forma sencilla y bastante potente de extraer información de las bases de datos puede que no sea la mejor para todo el mundo. Considero que para la visualización de los datos sería mucho más interesante conectar con una aplicación del estilo de Tableau. Y que para la monitorización del sistema sería más interesante el uso de una aplicación del estilo de Montecarlo, la cual nos permitirá observar la regularidad de los

accesos a la base de datos.

Resumen de configuraciones

En esta sección vamos a resumir las configuraciones adicionales que hay que efectuar para asegurar que el sistema funcione correctamente. La sección la dividiremos en tres

Librerías

Como se ha modificado bastante la funcionalidad del sistema debemos de asegurarnos de tener acceso a las siguientes librerias.

- nltk
- scikit-learn
- joblib
- numpy==1.20
- requests
- psycopg2-binary

IP

Para que el sistema funcione correctamente es necesario conocer la IP local del ordenador en el que se está ejecutando los contenedores docker, y de esta forma asegurar las comunicaciones entre ellos.

Los archivos donde será necesario de actualizar la IP serán:

- listener-service/kafka_consumer.py
- Database Access/Character DataSet.ipynb
- Database Access/Documents DataSet.ipynb
- Database Access/Join DataSet.ipynb

Activar listener

Y por último debemos de activar el listener para asegurar que los mensajes escritos por el módulo productor llegan a buen puerto.

Para levantarlo, una vez posicionado en la carpeta que contenga al "docker-compose.yml", debemos de ejecutar

- docker-compose exec listener-service bash

Y para arrancar el servicio

- python3 kafka_consumer.py