Pig.

Néstor Rodríguez Vico - nrv23@correo.ugr.es

11 de mayo de 2019

1. Origen de los datos.

El dataset empleado es un dataset sinteético montado con *Python* de forma aleatoria para representar la solución a un problema que cualquier empresa de desarrollo software puede tener. El dataset usado es sintético ya que no puedo usar datos reales de la empresa en la que trabajo por temas de confidencialidad, pero la esencia y la idea representada es la misma.

Cualquier producto software tiene un ciclo de desarrollo en el cual podemos, para este caso en particular, destacar dos etapas: el desarrollo del producto y el testeo del mismo. Cuando se trabaja en un software de gran tamaño y bastante complejo, la segunda etapa comentada es primordial.

A pesar de que el proceso de testeo de un producto es priomordial, es una tarea compleja, ya que hay que testear todas las partes de un software, las interconexiones de dichas partes y todas las casuísticas posibles, lo cúal hace dicha tarea bastante tediosa y compleja para ser realizada por un humano. Para ello, surgen los tests automátizados. Esto es, una técnica que permite, mediante código, crear tests para probar el producto software en cuestión.

Una vez conocemos esta posibilidad, surge la cuestión de como usarlos. La metodología a seguir cuando hay que modificar una pieza software es añadir una nueva funcionalidad o corregir una existente y programar los tests asociados a los cambios producidos. A continuación, se deben ejecutar todos los tests disponibles para asegurarnos que, los cambios introducidos, no han roto ninguna otra parte de nuestro código. Es decir, los tests existen para asegurarnos que, en un futuro, cuando hagamos cambios sobre nuestro software, el funcionamiento del mismo no ha cambiado.

Para automatizar aún más el proceso de testeo de un producto, surgen herramientas de Integración Continua (*Continuous Integration*) como pueden ser Travis CI o Jenkins. La idea de esta herramientas es, conectarlas a un repositorio de código y que cuando haya un *push* a ese repositorio, se ejecutan todos los tests disonibles generando una salida que indica como ha ido la ejecucion de los mismos. De esta manera, cuando un desarrollador quiere modificar un software, lo que debe hacer es añadir su código y sus tests, hacer un *push* al repositorio y esperar a que los tests hayan terminado de ejecutar en Travis CI o Jenkins para ver el resultado de los mismos.

Para una empresa u organización en la que se manejan varios productos software, es relevante tener un estudio de como han ido ejecutando tus tests a lo largo del tiempo, los resultados de los mismos. De esta necesidad, surge la base de datos empleada en esta práctica.

2. Estructura de los datos.

La base de datos elegida tiene las siguientes columnas:

- **id**. Tipo: *entero*. Representa un identificador único.
- **product**. Tipo: *texto*. Producto al que pertenece el test.

- **branch**. Tipo: *texto*. Rama del repositorio donde se encuentra el test ejecutado.
- **date**. Tipo: *texto*. Representa la fecha de ejecución de los tests.
- **run_number**. Tipo: *entero*. Representa el identificador Travis CI o Jenkins de la ejecución donde se han ejecutado los tests.
- **test_area**. Tipo: *texto*. Los tests pueden estar asociados a una área del producto que testean.
- **test_type**. Tipo: *texto*. Los testes pueden ser de interfaz (*UI test*), tests unitarios (*Unit Test*) o tests de sistema (*System Test.*).
- **class_name**. Tipo: *texto*. Clase a la que pertenece el tests.
- **method_name**. Tipo: *texto*. Nombre del método que ejecuta el test.
- **elapsed_time_sec**. Tipo: *número real*. Tiempo de ejecución del tests.
- **status**. Tipo: *texto*. Representa el resultado del test, puede ser *PASSDED*, *FAILED* o *ABORTED*.
- **error**. Tipo: *texto*. En el caso de que el test sea *FAILED* o *ABORTED*, tenemos almacenada información del porqué ha fallado el test.

3. Preparación del experimento.

Primero, debemos cargar los en *hdfs*. Para ello, ejecutamos el comando:

```
hdfs dfs -put tests.csv /user/impala/
```

A continuación, entramos a la shell de pig con:

```
pig
```

A continuación, cargamos los datos:

```
tests = load '/user/impala/tests.csv' using PigStorage(',') AS (
        id: int,
        product: chararray,
        branch: chararray,
        date: chararray,
        run_number: int,
        test_area: chararray,
        test_type: chararray,
        class_name: chararray,
        method_name: chararray,
        elapsed_time_sec: float,
        status: chararray,
        error: chararray
);
```

A continuación, para verificar que todo ha funcionado correctamente, vamos a mostrar los datos por pantalla

```
dump tests
```

4. Consultas.

4.1. Proyección.

A continuación, vamos a quedarnos la información relevante para nuestro problema:

```
informacion_relevante = FOREACH tests GENERATE product, branch, date, class_name,
    method_name, status;
```

4.2. Selección.

A continuación, quitamos las ejecuciones que son antiguas:

```
ultima_informacion_relevante = FILTER informacion_relevante BY date matches
    '.*2019-01-10 .*';
```

4.3. Agrupamiento y resúmenes de información.

Finalmente, calculamos el número de tests fallados, el número de tests totales y porcentaje de tests fallados en cada rama de cada producto:

```
data_grouped = GROUP ultima_informacion_relevante BY (product, branch);

solucion = FOREACH data_grouped {
        failed_bag = FILTER ultima_informacion_relevante BY status == 'FAILED';
        GENERATE group, COUNT(failed_bag) as failed_counter,
        COUNT(ultima_informacion_relevante) as total_counter,
        (double) COUNT(failed_bag) * 100.0 / (double) COUNT(
            ultima_informacion_relevante);
};

dump solucion;
```