Práctica Copia.me

Tomado del contexto del final 22/12/2018

Contexto

Copia.me será un sistema en el que los usuarios subirán documentos de distintos tipos con el fin de analizarlos para detectar si estos son plagios.

Por ahora soportaremos documentos de texto, programas y partituras pero se espera incorporar más. También se contará con revisores freelancer que se encargarán de analizar manualmente los documentos para darles un análisis más inteligente.

Copia.me estará orientado a docentes, que cargarán lotes de documentos y recibirán un reporte que indique cuáles son copias entre sí.

Los usuarios de Copia.me podrán elegir entre tres calidades de servicio distintas; cada una realiza distintos análisis y por supuesto esto afectará en el precio de la revisión. Las calidades de servicios con los que contamos son:

● Bronce: sólo realiza detección automática, usando diferentes algoritmos, dependiendo del tipo de contenido:
    ○ Los documentos legales, literarios y académicos se comparan usando la distancia de Levenshtein.
    ○ Los fragmentos de código se analizan usando Detección de clones basado en árboles sintácticos.
    ○ Las partituras musicales se analizan usando Acustic fingerprint.

● Plata: además de la detección automática, envía un cierto porcentaje (configurable por el usuario, esto influencia en el costo) de los documentos a analizar contra revisores freelancers que realizan una revisión manual. Cada revisor recibe un email con cada par de documentos asignados.

● Oro: además de la detección automática, envía todos los documentos a los revisores, y además hace una revisión cruzada: un cierto porcentaje de los mismos (nuevamente, configurable por el usuario, esto influencia en el costo) los envía a más de un revisor.

Cuando un análisis (detección automática, revisión manual o revisión cruzada) termina, genera un índice de copia para cada par de documentos, entre 0 y 1. Y cuando todos los análisis correspondientes para una calidad de servicio terminan, se marca a cada documento afectado como revisado. Se considera plagio cuando el documento recibe un promedio de puntaje mayor a 0,6. Tener en cuenta que estos análisis son asincrónicos, por lo que pueden terminar en cualquier orden.

Obviamente, contar con un gran equipo de revisores freelancers no es tarea fácil, así vamos a asumir algunas cosas:

1. De cada freelancer sabemos cuántos documentos puede revisar por máximo cada mes, y su email.
2. Siempre habrá revisores disponibles.
3. Los revisores ingresarán al sistema y cargarán los resultados de sus análisis, siempre en el plazo máximo de 1 día.
4. Para la comunicación con los revisores, ya contamos con un componente que nos permite enviar emails usando SMTP (aunque en un futuro tal vez contratemos un servicio externo).

Repo Resuelto: https://github.com/ezequieljsosa/ddscopiame

1ra Iteración

El objetivo de la primera iteración es familiarizarse con la tecnología Java, Maven y de testing. Por otro lado hacer una primera aplicación, siguiendo las capas aprendidas en clase.

El alcance será el siguiente:

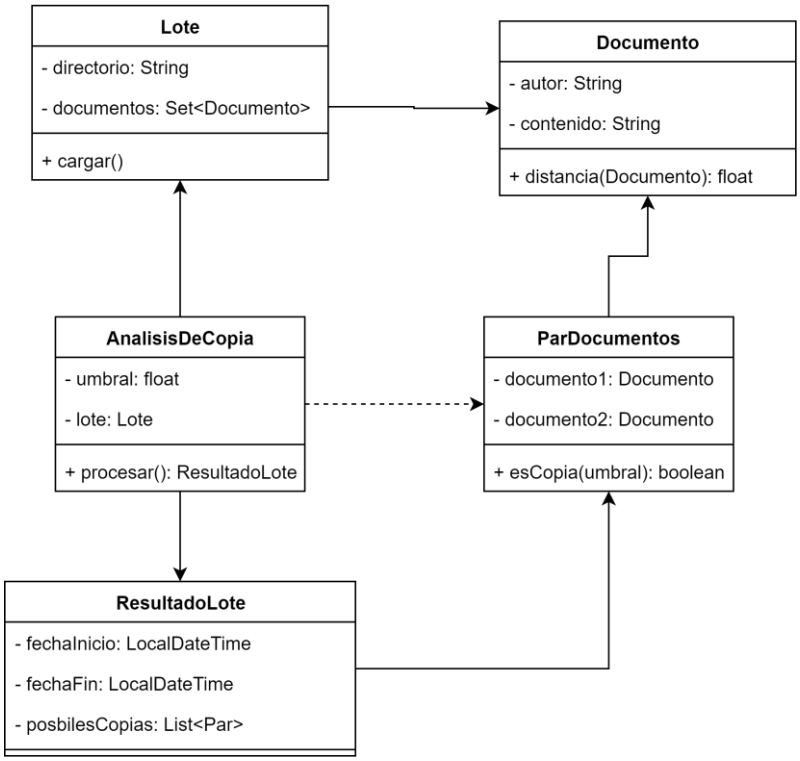
- Solo detección automática de documentos txt.
- Proceso monousuario / “monodocente”
- El lote de documentos, será procesado localmente, mediante un programa de línea de comandos.

Conceptos importantes antes de iniciar:

- Java es un lenguaje fuertemente tipado y compilado. No se darán mayores explicaciones de la sintaxis por su parecido con Wollok. Cualquier duda puede consultar los apuntes asociados
- JUnit: framework de testeo unitario para Java. Permite definir fixture de pruebas y obtener reportes sobre los resultados de las mismas
- Maven es una herramienta para la construcción y creación de proyectos, de difundido uso en Java. Tiene numerosas características, pero nosotros resaltaremos las siguientes:
    - Propone una estructura de directorios para ordenar el código
    - Nos ayuda a gestionar las dependencias
    - Ofrece un formato para configurar el proyecto en un archivo aparte

Paso 1: Diseño

Se propone el siguiente diseño para esta iteración:



1. Estamos suponiendo que los documentos no tienen un gran tamaño, por eso leemos su contenido completo
2. Comparar 2 documentos implica tener algún concepto de distancia, en el alcance de esta iteración la Levenshtein.
3. Se guardan los pares, para tener registro de cuales son copia
4. Como veremos más adelante, cuando se procesa un conjunto de datos, a parte de los resultados, es conveniente guardar el inicio y final de dicho proceso
5. El umbral del análisis determina si 2 documentos son copia por arriba de ese valor (en principio). Como todos los documentos de un lote deben ser procesados con el mismo criterio, el mismo esta en el análisis
6. ParDocumentos#esCopia(umbral): documento1.distancia(documento2) < umbral

Preguntas para hacerse hasta el momento

- Relacione el 1er punto con algún tema visto en la teórica
  Esta relacionado con los recursos que se estiman que van a ser necesarios.
- ¿Por qué le parece que el umbral está en el AnalsisDeCopia y no en Lote?
  Para que lote sea mas cohesivo y poder abstraer responsabilidad responsabilidad .
- ¿ Que opina que el umbral sea un parámetro y no un valor fijo? Por ejemplo ParDocumentos#esCopia(umbral): documento1.distancia(documento2) < 0.6
  Me parece una correcta práctica, esto da la posibilidad que el día de mañana pueda ser editado por el usuario sin necesidad de tocar el código fuente.

Paso 2: Creación del proyecto

La forma más fácil de crear un proyecto Maven es a través de la IDE[1]. Por ejemplo en Eclipse la mejor forma File -> New -> Maven Proyect, chequear “Create a simple project”, siguiente -> Group Id: ar.edu.utn.dds.copiame ; ArtifactId copiame-cli ; => finalizar.

Confiamos en la sapiencia y criterio de un alumno de 3er año para realizar los pasos anteriores en otras IDEs y cualquier cambio que haya que hacer para que funcione.

Con esto vamos a tener la siguiente estructura de proyecto

src
  main
    java : en este directorio van los paquetes / clases de mi programa
    resources : cualquier recurso que no sea código utilizado por las clases / app (imágenes, sonidos, texto, etc..)
  test
    java : Clases de para hacer pruebas: Fixtures y clases auxiliares
    resources : cualquier archivo que se utilice para realizar pruebas, y/o sets de datos

pom.xml : Nombre y configuración del proyecto, independiente de la IDE o ambiente de desarrollo. Por ejemplo versión de Java y dependencias.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>ar.utn.dds.copiame</groupId>
	<artifactId>copiame-cli</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<!-- Todo lo que esta de aquí en adelante hay que agregarlo   -->
	<properties>
		<maven.compiler.target>17</maven.compiler.target>
		<maven.compiler.source>17</maven.compiler.source>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.junit.jupiter</groupId>
			<artifactId>junit-jupiter-api</artifactId>
			<version>5.9.0</version>
			<scope>test</scope>
		</dependency>
	<!-- Para el cálculo de la distancia de Levenshtein   -->
		<dependency>
		    <groupId>org.apache.commons</groupId>
		    <artifactId>commons-text</artifactId>
		    <version>1.10.0</version>
		</dependency>
	</dependencies>
</project>
```

[1] si bien por lo general recomendamos usar más la línea de comandos, para una primera práctica usar este método está perfecto

Tips: verificar que tienen la misma versión de Java instalada (Recomendamos fuertemente usar la 17). Se elige encoding se pone el más estándar por las dudas.

En este punto es conveniente agregar al proyecto, al archivo .gitignore en la raíz del mismo (a la misma altura que el pom.xml). Para esto recomendamos usar este sitio, y colocar los tags: Maven, Java, Eclipse, Intellij+iml. Esto nos ayudará a no hacer commit de archivos que no quiero versionar. Más adelante vamos a nombrar un par de ejemplos

Paso 3: Implementación del dominio

En este punto, implemente todas las clases, menos los métodos públicos: Lote#cargar, Documento#distancia, AnalisisDeCopia#procesar, ParDocumentos#esCopia (déjelos a todos retornando null). Todo sobre src/main/java y recuerden poner un paquete, por ejemplo “ar.utn.dds.copiame”. Los paquetes siempre van en minúscula y con el formato del ejemplo que se dio. Ejemplo para ResultadoLote:

```
package ar.utn.dds.copiame; // Las clases SIEMPRE deben estar dentro de un
paquete
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class ResultadoLote {
	private LocalDateTime fechaInicio;
	private LocalDateTime fechaFin;
	private List<ParDocumentos> posiblesCopias;

	public ResultadoLote() {
		super();
		this.posiblesCopias = new ArrayList<ParDocumentos>();
	}
	public LocalDateTime getFechaInicio() {
		return fechaInicio;
	}
	public void setFechaInicio(LocalDateTime fechaInicio) {
		this.fechaInicio = fechaInicio;
	}
	public LocalDateTime getFechaFin() {
		return fechaFin;
	}
	public void setFechaFin(LocalDateTime fechaFin) {
		this.fechaFin = fechaFin;
	}
	public List<ParDocumentos> getPosiblesCopias() {
		// encapsulamos la colección para que no puedan manipularla sin
	usar agregarPar
		return new ArrayList<ParDocumentos>(posiblesCopias);
	}
	public void agregarPar(ParDocumentos par) {
		this.posiblesCopias.add(par);
	}
}
```

Paso 4: Testing

Ahora vamos a implementar un par de pruebas unitarias, las cuales consisten en clases con una estructura en particular, para que puedan ser interpretadas por el framework JUnit. Dichas clases se crean sobre src/test/java, sobre el MISMO paquete de las clases que prueban. Por ejemplo vamos a construir el fixture de DocumentoTest, es decir la clase que tendrá las pruebas de la clase Documento.

```
package ar.utn.dds.copiame;
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class DocumentoTest {
	@Test
	public void testDocumentosIguales() {
		Documento doc1 = new Documento("","hola");
		Documento doc2 = new Documento("","hola");
		assertEquals(0,doc1.distancia(doc2),
			"los textos que son iguales tienen que dar una distancia de
		0");
	}
	@Test
	public void testDocumentosDistintos() {
		Documento doc1 = new Documento("","chau");
		Documento doc2 = new Documento("","hola");
		assertEquals(1,doc1.distancia(doc2),
			"los textos que son totalmente distintos tienen que dar una
		distancia de 1");
	}
	@Test
	public void testDocumentosParecidos() {
		Documento doc1 = new Documento("","hola");
		Documento doc2 = new Documento("","halo");
		// el 3er parámetro es una toleracia para comparar (decir si son
		iguales) los números con punto flotante
		assertEquals(0.5,doc1.distancia(doc2),0.1,
			"los textos que son parecidos en un 50% tienen que dar ~0.5");
	}
}
```

Si ejecutamos las pruebas fallaran, todas si pusimos que “distancia” retorne null, o 2 si pusimos que retorne “0”, pero esa esta funcionando de casualidad. Ahora vamos a implementar el método de distancia:

```
public Float distancia(Documento otroDoc) {
	/* La distancia de Levenshtein mide la cantidad de operaciones
	* (inserción, eliminación o sustitución de un carácter) necesarias
	para transformar un texto en otro */
	// Mientras más distintos los textos, más alta es la distancia
	Integer rawDist = LevenshteinDistance.getDefaultInstance(
		).apply(this.contenido, otroDoc.getContenido());
	// Normalizamos por el tamaño de la distancia por el tamaño del texto
	mas largo
	Integer contentSize = Math.max(this.contenido.length(),
	otroDoc.getContenido().length());
```

```
        return rawDist * 1.0f / contentSize;
        }
```

Ahora las pruebas deberían funcionar normalmente.

Paso 5: Actividad Test

Realice 2 pruebas sobre la clase ParDocumentos. ¿ Qué método es el más relevante? ¿ Qué situaciones se le ocurre probar ? ¿ Hace falta más de 2 tests ?

El método mas relevante es "esCopia()". Es necesario probar el caso cuando dos elementos son una copia, el caso en el que no y en el caso que la distancia de igual que el umbral definido. En total serían necesarios 3 Test para este método.

Paso 6: Implementación de la Aplicación

Nuestra aplicación, hasta el momento, es un CLI (línea de comandos). Por ello, la interacción con el usuario se puede dar de las siguientes formas:

1. Con un argumento cuando se ejecuta el programa.
2. Pidiendo una interacción al usuario.
3. Muestra datos a través de la consola.
4. Puede escribir algo en uno o más archivos, u otros lugares.
5. Interactuar con otros sistemas.

Para que sea más fácil, nosotros utilizaremos (1) y (3). Es decir, no habrá ningún pedido de datos al usuario una vez que se arrancó el programa, este simplemente indicará qué información quiere procesar y obtendrá la respuesta por consola.

Antes que nada, tenemos que establecer pre-condiciones, sobre las entradas y salidas del sistema:

- Entrada: Directorio con archivos txt, donde el nombre de cada archivo es el autor del mismo.
- Salida: personas que probablemente se hayan copiado entre ellas

```
public class CopiaMeApp {
	public static void main(String[] args) {
		// Valido argumentos del usuario --> Capa ?
		Path pathLote = Paths.get(args[0]);
		if ( ! Files.exists( pathLote )) {
			System.err.println("'" + args[0] + "' no existe...");
			System.exit(1);
		}
		// ------------------------------------
		//Cargo el Lote del Sistema de archivos --> Utilizo la Capa ?
		Lote lote = new Lote(args[0]);
		lote.cargar();
		// ------------------------------------------------
		// Utilizo la capa de ? -- NO leo datos de otra fuente -- NO
		pido ni muestro información
		float umbral = 0.5f;
		AnalsisDeCopia analisis = new AnalsisDeCopia(umbral, lote);
		ResultadoLote resultado = analisis.procesar();
		//--------------------
```

```
            // Muestro la informacion por pantalla --> Capa de ?
            for (ParDocumentos par : resultado.getPosiblesCopias()) {
                        System.out.println(par.getDocumento1().getAutor() +
                  " " + par.getDocumento2().getAutor() );
            }
            //-------------
      }}
```

Método procesar de AnalsisDeCopia

Primero agreguemos una libreria que nos permitira hacer las combinaciones de los documentos:

```
<!-- https://mvnrepository.com/artifact/com.github.dpaukov/combinatoricslib3 -->
            <dependency>
                  <groupId>com.github.dpaukov</groupId>
                  <artifactId>combinatoricslib3</artifactId>
                  <version>3.3.3</version>
            </dependency>
```

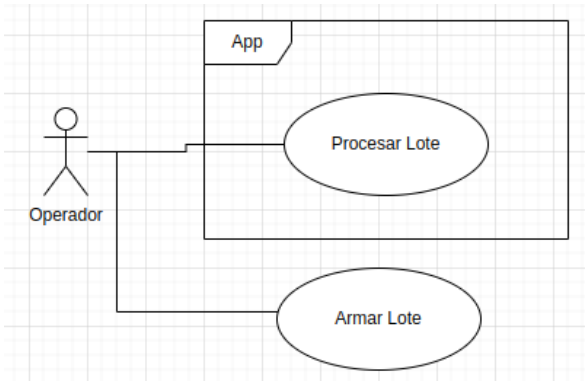
```
public ResultadoLote procesar() {
      // Genero todos los pares de documentos Posibles
      List<ParDocumentos> pares =
      Generator.combination(this.lote.getDocumentos())
            .simple(2)
            .stream()
            .map(t-> new ParDocumentos(t.get(0),t.get(1)) )
            .collect(Collectors.toList());
      // Armo el resultado procesando cada par
      ResultadoLote rl = new ResultadoLote();
      rl.setFechaInicio(LocalDateTime.now());
      for (ParDocumentos parDocumentos : pares) {
            if(parDocumentos.esCopia(this.umbral)) {
                  rl.agregarPar(parDocumentos);
            }
      }
      rl.setFechaFin(LocalDateTime.now());
      return rl;
}
```

Actividades:

1) Complete las capas en los comentarios del método `CopiaMeApp#main`
2) Vea el diagrama UML, por que cree que la linea es distinta de AnalsisDeCopia a ParDocumentos respecto a ResultadoLote

Paso 7: Capa fuente de datos

Para arrancar, tenemos que poner unas precondiciones. Analicemos los CU que tenemos hasta el momento.



Tenemos 2 cosas para marcar, primero, armar el lote, si bien es necesario, no ofrecemos funcionalidad para hacerlo, eso tiene que hacerlo el actor por fuera de la aplicación. Por otro lado, debemos ser precisos a la hora de definirlo, ya que esas son las precondiciones del CU Procesar Lote:

- Toma de entrada un directorio
- Dentro del mismo hay archivos .txt, los que no lo sean, serán ignorados para la carga
- El nombre del archivo será el nombre del autor del documento (otra posibilidad seria tener el vínculo en un archivo aparte, pero vamos a hacerlo de esta forma). Para evitar algunos problemas, los espacios serán separados por "_"

Luego podemos plantear escenarios de error, tales como: el directorio no existe, el directorio tiene menos de 2 documentos, ¿qué hay archivos vacíos?, ¿qué hacer si se detecta que un archivo no es de texto a pesar de ser .txt ?. Por ahora solo trataremos la situación de que el directorio no exista.

Programe Lote#cargar, puede usar los siguientes métodos para ayudarse:

```
String directorio = "una/ruta"
Path dirpath = Paths.get(directorio);
Files.list(dirpath).collect(Collectors.toList()) ; // nos da los archivos de un directorio (suponiendo que dirpath es un directorio)
A los objetos de la clase Path le puedo pedir .getFileName(), para que nos dé el nombre de archivo completo
```

Paso 8: Finalice el programa

Complete lo que haga falta y pruebe el main, con un par de lotes distintos.

- ¿Cree usted que falta interacción con el usuario?

Personalmente creo que se podría sumar que el ingreso de la ruta del directorio sea por una interacción en vez del ingreso por argumento.

- ¿Si tuviese que dar la opción de guardar el resultado en un archivo, donde lo haría?

En esta situación que no se está trabajando con un D se podría generar un archivo txt con el resultado dado.