Práctica 1: Preprocesado de documentos

Juan Manuel Consigliere Picco, Óscar Pérez Tobarra

1.Código

Ejecución: java -cp tika-app-1.24.jar:. Practica1 directorio [-d|-l|-t] 2> /dev/null

Se le añade “2> /dev/null” al final para redirigir los warnings que emite Tika al ejecutar el programa y que no se impriman a la vez que las salidas deseadas.

```
Tika tika = new Tika();
Metadata metadata = new Metadata();
File dir = new File(args[0]);
File[] archivos = dir.listFiles();

if(args.length != 2){
  System.out.println("Número incorrecto de parámetros de entrada.");
  System.exit(0);
}
```

Figura 1: inicialización del programa.

Como se ve en la figura 1, el programa comienza creando una instancia de Tika, una objeto Metadata para almacenar los metadatos de los archivos, un objeto File que almacena el directorio que se le pasa como parámetro por consola, y un array de objetos File que almacena los ficheros que se encuentran en el directorio ya mencionado. También incluye una comprobación para ver si se le han pasado todos los argumentos necesarios para la ejecución (el directorio y la opción de ejecución).

- **Opción -d**

```
if(args[1].equals("-d")){
  System.out.format("| Título                                | Tipo de fichero           | Codificacion            | Idioma             |%n"+
                    "|---------------------------------------|---------------------------|-------------------------|--------------------|%n");

  for (File archivo : archivos) {
    ArrayList<String> info = new ArrayList<>(); //Estructura para almacenar los datos que se buscan
    info.add(archivo.getName()); //Obtenemos el nombre del fichero
    info.add(tika.detect(archivo)); //Obtenemos el tipo de fichero

    String text = tika.parseToString(archivo); //Se parsea el fichero a texto plano
    tika.parse(archivo,metadata); //Parseamos el fichero de texto plano

    info.add(detectarLenguaje(text)); //Detectamos el lenguaje escrito del documento (implementado arriba)

    /*
    Obtenemos la codificación a partir de los metadatos ó, en caso de que no se incluya la codificación en los metadatos, con
    la función detectarCodificación() (implementada arriba)
    */
    if(metadata.get(Metadata.CONTENT_ENCODING) != null ){
      info.add(metadata.get(Metadata.CONTENT_ENCODING));
    } else{
      info.add(detectarCodificacion(text));
    }

    String formato = "| %-36s   |  %-25s | %-23s | %-18s |%n";
    System.out.format(formato, info.get(0), info.get(1), info.get(3), info.get(2));
  }
}
```

Figura 2: opción -d

Esta opción devuelve al ejecutarse el nombre, el tipo (extensión), la codificación y el idioma de cada uno de los ficheros contenidos en el directorio proporcionado a través de la consola. Vamos a explicar la funcionalidad integrada.

Para cada objeto File incluido en el array de ficheros, se crea un ArrayList de String para almacenar los datos que se buscan. Después, obtenemos del objeto File el nombre y el tipo de fichero.

Para obtener la codificación y el idioma tenemos que parsear primero el documento a String, y luego parsearlo con Tika para obtener los metadatos. Una vez hecho ya podemos consultar los metadatos para obtener el idioma y la codificación.

```
public static String detectarLenguaje (String text) {
    LanguageDetector identifier = new OptimaizeLangDetector().loadModels();
    LanguageResult idioma = identifier.detect(text);
    return idioma.getLanguage();
}

public static String detectarCodificacion (String text) {
    CharsetDetector detector = new CharsetDetector();
    detector.setText(text.getBytes());
    CharsetMatch encoding = detector.detect();
    return encoding.getName();
}
```

Figura 3: funciones detectarLenguaje() y detectarCodificacion()

El idioma se obtiene mediante la función detectarLenguaje() creada por nosotros. Esta función utiliza el método detect de la clase LanguageDetector para identificar el idioma en el se escribe, devolviendo como resultado un identificador de dos letras del idioma (es-español, en-inglés, etc.).

En el caso de la codificación se puede obtener del campo de los metadatos CONTENT_ENCODING, pero hay un problema: los ficheros .pdf no incluyen la codificación en sus metadatos, lo cual da pie a pensar que existen otros formatos que tampoco lo incluyen. Aún así, existe una forma alternativa de obtener la codificación: usando el método detect() de la clase CharsetDetector, al cual le pasamos el texto parseado en un array de bytes. Así, planteamos lo siguiente: si el campo CONTENT_ENCODING es nulo o vacío, llamamos al método detectarCodificación(); en caso contrario simplemente cogemos el valor almacenado en ese campo.

Una vez hecho, solo resta mostrar los datos obtenidos en una tabla impresa por consola. Se ha decidido usar el método format(), el cual nos permite determinar la forma en la que se imprimen los resultados en pantalla. En la figura 4 se ve el resultado de la ejecución de esta opción.

```
[ÓscarPérezTobarra opereztobarra@DESKTOP-FP386RF:/mnt/c/Users/Usuario/Desktop/Cosas/Documentos/4º/RI/Practica 1] 2021-10
-08 Friday
$java -cp tika-app-1.24.jar:. Practica1 ./test -d 2> /dev/null
| Título                              | Tipo de fichero            | Codificacion            | Idioma             |
|-------------------------------------|----------------------------|-------------------------|--------------------|
| alykkaan_ritarin_don_quijote.txt    |  text/plain                | UTF-8                   | fi                 |
| balazs_sandor_beszelyei.epub        |  application/epub+zip      | UTF-8                   | hu                 |
| die_verwandlung.txt                 |  text/plain                | UTF-8                   | de                 |
| divina_commedia.epub                |  application/epub+zip      | UTF-8                   | it                 |
| germana.epub                        |  application/epub+zip      | UTF-8                   | es                 |
| guion_p1.pdf                        |  application/pdf           | UTF-8                   | ca                 |
| hamlet.pdf                          |  application/pdf           | UTF-8                   | es                 |
| le_chevalier_deon.epub              |  application/epub+zip      | UTF-8                   | fr                 |
| loremipsum.txt                      |  text/plain                | UTF-8                   | ca                 |
| platero_y_yo.pdf                    |  application/pdf           | UTF-8                   | es                 |
```

Figura 4: resultado de la ejecución -d en un directorio /test con 10 ficheros

- **Opción -l**

Para este apartado convertimos los archivos que recorrimos a tipo InputStream, que es el formato que admite la función parse de AutoDetectParser. El cometido de crear un objeto AutoDetectParser es detectar el tipo de documento que está leyendo y parsearlo de la manera adecuada. Esto nos facilita mucho la implementación.

Para lo anterior creamos varios objetos tipo LinkContentHandler (estructura de almacenamiento de enlaces), ParseContext (contexto) y Metadata, que ya declaramos anteriormente, ya que los utilizamos en varias ocasiones, y finalmente todos esos objetos se los pasamos como argumento a la función parse. Finalmente añadimos todos los links en formato Link a una lista.

```
InputStream input = new FileInputStream(archivo); // Convertimo
LinkContentHandler link = new LinkContentHandler(); // Estructu
ParseContext contexto = new ParseContext(); // Indica el contex
AutoDetectParser parser = new AutoDetectParser(); // Detecta el

parser.parse(input, link, metadata, contexto); // Lee el docume
List<Link> links = link.getLinks();
```

Figura 5: argumentos de AutoDetectParser y parseo de documento

A la hora de imprimir el resultado nos dimos cuenta que para algunos archivos había objetos de la lista que estaban vacíos, o realmente no eran enlaces. Esto último ocurría porque detectaba los hipervínculos para redirigirte a la información de pie de página.

Finalmente, ya que no nos servía para nada decidimos añadir una condición que detectara los objetos que contengan enlaces al pie de página (“@”) y los vacíos (“”), y no los imprimieramos. Para ello debíamos pasar cada Link a String con la función toString, como se ve en la captura siguiente:

```
System.out.println("Archivo: "+archivo.getName());

if(links.isEmpty())
  System.out.println("No se han encontrado enlaces.");
else{
  j = false;
  for(Link i : links){
    if(!i.getUri().toString().contains("@") && !i.getUri().toString().equals("")){
      System.out.println("\t"+i.getUri());
      j = true;
    }
  }

  if(!j)
    System.out.println("No se han encontrado enlaces.");
}
```

Figura 6: proceso de impresión del resultado final

También hemos contemplado la posibilidad de que no hayan enlaces en un archivo, así que cuando esto ocurre el programa nos notifica correctamente de la situación mediante la condición de si la lista está vacía, y si no lo está pero sus enlaces no son válidos tenemos un booleano que nos avisa, y si el booleano nunca se ha activado significa que ningún enlace era válido, por lo que nos lo notifica. El nombre del archivo lo sacamos de la misma manera que en la opción -d. Un ejemplo de ejecución de este argumento es el siguiente:

```
jcpicco@DumbleDogg:/mnt/c/Users/jmcon/Documents/GitHub/ri/p1$ java -cp tika-app-1.24.jar:. Practica1 ./test/ -l 2>/dev/n
ull
Archivo: alykkaan_ritarin_don_quijote.txt
No se han encontrado enlaces.

Archivo: balazs_sandor_beszelyei.epub
        https://www.gutenberg.org
        https://www.gutenberg.org
        https://www.gutenberg.org/donate/
        https://www.gutenberg.org

Archivo: die_verwandlung.txt
No se han encontrado enlaces.

Archivo: divina_commedia.epub
        https://www.gutenberg.org
        https://www.gutenberg.org
        https://www.gutenberg.org/donate/
        https://www.gutenberg.org

Archivo: germana.epub
No se han encontrado enlaces.

Archivo: guion_p1.pdf
        https://tika.apache.org/
        https://tika.apache.org/
```

Figura 7: resultado de la ejecución -l en un directorio /test con 10 ficheros

- **Opción -t**

Primero añadimos a un String el archivo parseado, y pasado completamente a minúsculas. Para ello utilizamos la función de Tika parseToString(), acompañada de un toLowerCase().

Después utilizamos la función split() al texto anterior para separar las palabras con caracteres raros, y nos devuelve un array de String. Esta función tiene una documentación pésima y nos costó bastante comprender cómo funcionaba, pero tras prueba y error y tras muchas ejecuciones conseguimos seleccionar la mayoría de los caracteres raros para separar correctamente cada palabra.

```
for (File archivo : archivos) {
  String text = tika.parseToString(archivo).toLowerCase(); //Se parsea el fichero a texto plano

  /*
  Split: Añade a un array de String las palabras, separándolas por los caracteres que hemos especificado.
  */
  String[] split = text.split("\\s+|\\.|\\,|\\;|\\?|\\!|\\¿|\\¡|\\(|\\)|\\{|\\}|\\[|\\]|\\:|\\|"+
                              "|\\-|\\-|\\-|\\+|\\_|\\<|\\>|\\/|\\=|\"|\'|\\#|\\$|\\%|\\–|\\\\|\\•|\\’");
```

Figura 8: parseado a string y split() por carácteres especiales

Para las ocurrencias hemos decidido utilizar un HashMap, con clave String y valores Integer. Recorremos el array de String anterior inicializando un contador a 0 por palabra, y miramos si la palabra ya se encuentra en el HashMap. Si no se encuentra recorremos el String en un bucle for otra vez, y cuando encontremos la misma palabra aumentamos en 1 el contador. Finalmente al terminar el bucle añadimos un objeto al HashMap con clave la palabra, y valor el contador.

```
Map<String, Integer> ocurrencias = new HashMap<String, Integer>();

for(String palabra: split){
  int contador = 0;
  if(!ocurrencias.containsKey(palabra)){
    for(int i = 0; i < split.length; i++){
      if(palabra.equals(split[i]))
        contador++;
    }

    ocurrencias.put(palabra, contador);
  }
}
```

Figura 9: conteo de palabras del array de String y relleno de HashMap

En el enunciado se nos pide ordenar las palabras por su número de apariciones en el texto, de mayor a menor. Hemos pasado el HashMap a una lista con la clase abstracta Collectors. Luego hemos usado la función Sort de la clase Collections, la cual se le pasa la lista anterior y se le pasa un objeto Comparator, al cual se le debe especificar cómo va a comparar los objetos de la lista, y si no se especifica no funciona (por ser una clase abstracta).

```
List<Map.Entry<String, Integer>> words = ocurrencias.entrySet().stream().collect(Collectors.toList());
Collections.sort(words, new Comparator<Map.Entry<String, Integer>>(){
  public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2){
    return o2.getValue().compareTo(o1.getValue());
  }
});
```

Figura 10: conversión HashMap-Lista y comparado de objetos para ordenar

Nos dimos cuenta que en la lista había aún valores numéricos y, por alguna razón, valores vacíos. Los valores numéricos no pudimos tratarlos con split(), y los vacíos posiblemente sean cuando hay varios símbolos raros seguidos, que intenta tomar lo siguiente que ve y no hay nada. Esto lo hemos arreglado añadiendo a un ArrayList todos los objetos de la lista que sean numéricos o vacíos. Hemos utilizado matches para la búsqueda de números, ya que con ella puedes utilizar expresiones regulares, y equals() para los vacíos.

En cuanto tenemos todas las palabras que queremos borrar, podemos recorrer el ArrayList creado y borrar cada objeto de la lista original que coincida con el del ArrayList. Así nos quedamos solamente con las palabras que nos interesan.

```
ArrayList<Map.Entry<String, Integer>> aborrar = new ArrayList<>();

for(Map.Entry<String, Integer> i : words){
  if(i.getKey().matches(".*\\d.*") || i.getKey().equals("")){
    aborrar.add(i);
  }
}

for(Map.Entry<String, Integer> i : aborrar){
  words.remove(i);
}
```

Figura 11: borrado de números y vacíos

De la impresión del resultado no hay mucho que decir. Utilizamos el formato que se nos especificó en el guión de la Práctica, así que mucho más no hay para decir. El nombre del archivo se saca como en la opción -d (con archivo.getName()).

Ahora procedemos a añadir el resultado de la impresión a un archivo .csv. Creamos un archivo tipo PrintWriter para escribir dentro de un archivo, y ahí podemos especificar la ruta y extensión en un solo argumento de String (especificando un nuevo archivo File). Para no tomar la extensión del archivo original, tomamos un substring del nombre hasta la posición del “.” (con substring() y lastIndexOf(), donde se especifica el carácter a buscar).

Recorremos la lista de palabras que teníamos y con la función write() del objeto PrintWriter, escribimos la clave seguida de un “;” y el valor. La primera línea que escribimos es “Text;Size”. Finalmente hacemos un close() para cerrar el archivo y escribir el resultado.

```
System.out.println();

PrintWriter writer = new PrintWriter(new File("./csv/"+archivo.getName().substring(0, archivo.getName().lastIndexOf("."))+".csv"));
writer.write("Text;Size\n");

for(Map.Entry<String, Integer> i : words){
  writer.write(i.getKey()+";"+i.getValue()+"\n");
}

writer.close();
```

Figura 12: escritura de fichero .csv

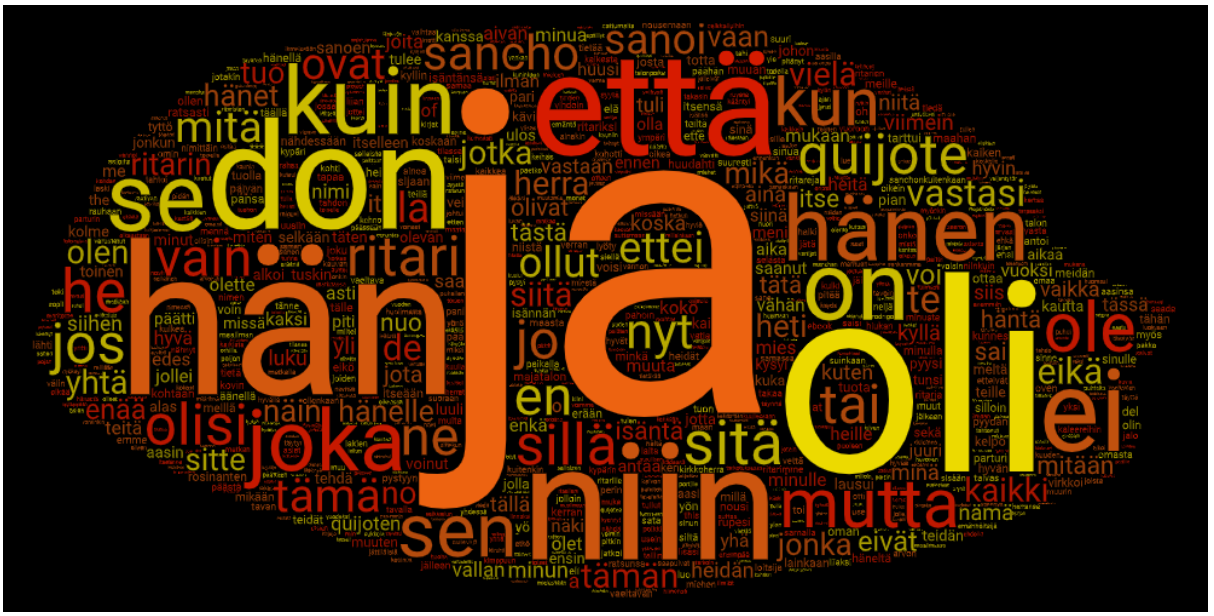
- **¿Qué ocurre si no introducimos las opciones disponibles?**
  Hemos puesto un mensaje de error, que nos dice que que la opción no es válida, y nos muestra las opciones de uso:

```
else{
  System.out.println("Error, opción introducida no válida.");
  System.out.println("Opciones de uso válidas:");
  System.out.println("-d: muestra el título, el tipo de archivo, la codificación y el idioma de los archivos del directorio proporcionado.");
  System.out.println("-l: muestra todos los enlaces extraíbles de los documentos pertenecientes al directorio proporcionado.");
  System.out.println("-t: genera un archivo CSV con las ocurrencias de cada término en los documentos del directorio proporcionado.");
}
```

Figura 13: mensaje de error por argumentos no válidos

2.Nubes de palabras

- **alykkaan_ritarin_don_quijote.txt**



- **balazs_sandor_beszelyei.epub**

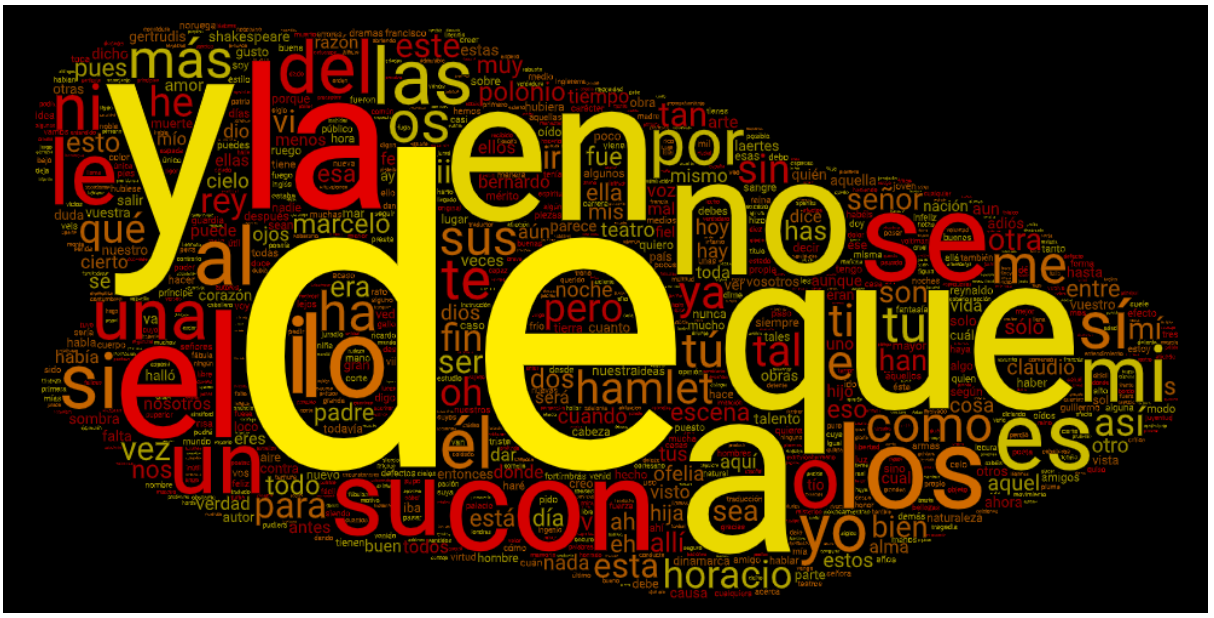
- **die_verwandlung.txt**

- **divina_commedia.epub**

- **germana.epub**

- **guion_p1.pdf**

- **hamlet.pdf**



- **le_chevalier_deon.epub**

- **loremipsum.txt**

- **platero_y_yo.pdf**

3.¿Se cumple la Ley de Zipf?

Para comprobar que se cumple la ley de Zipf vamos a coger tres documentos, cada uno con un idioma diferente, y vamos a generar las gráficas a partir de los archivos .csv generados por el programa.

- Divina Commedia
  Gráfico de ocurrencias:

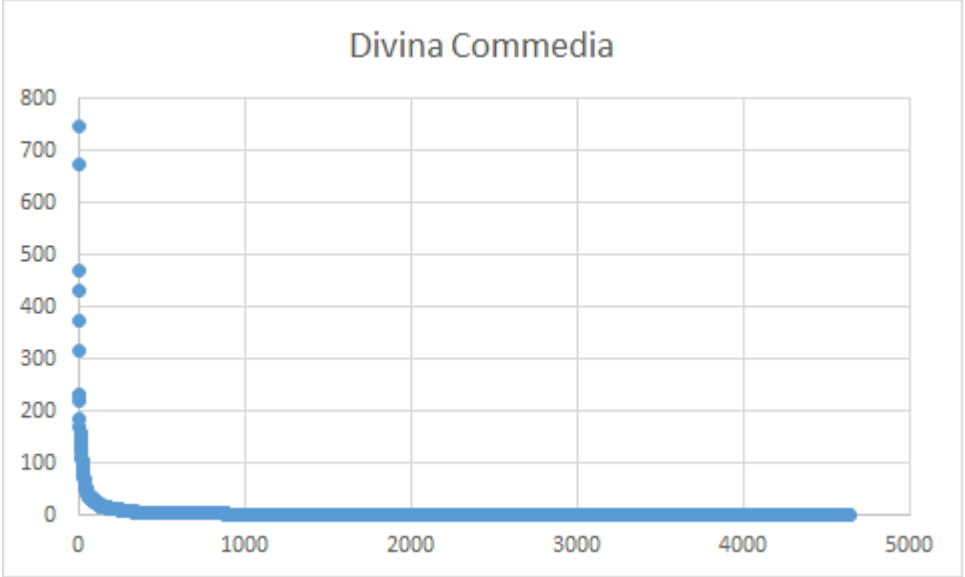
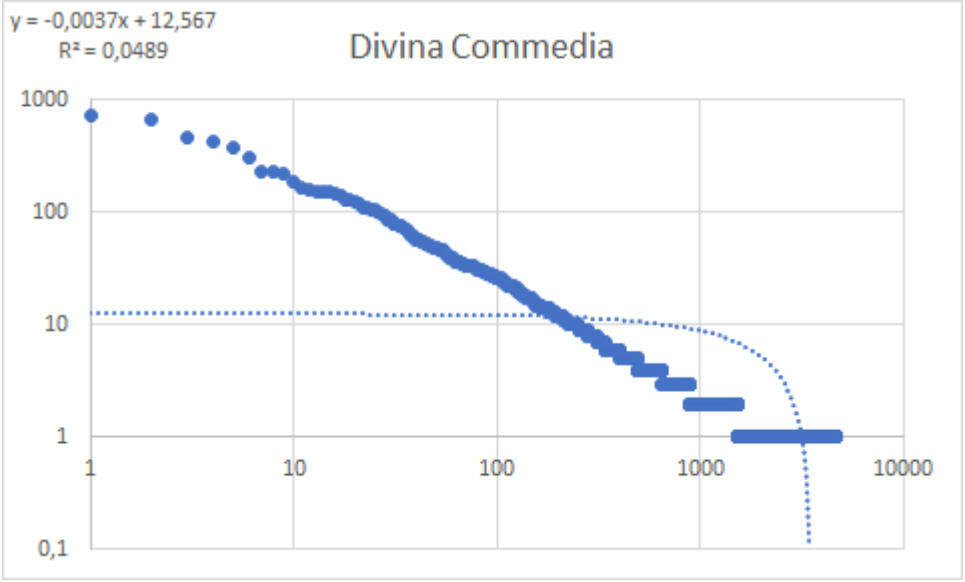


Gráfico log-log:



- Le Chevalier D’Eon
  Gráfico de ocurrencias:

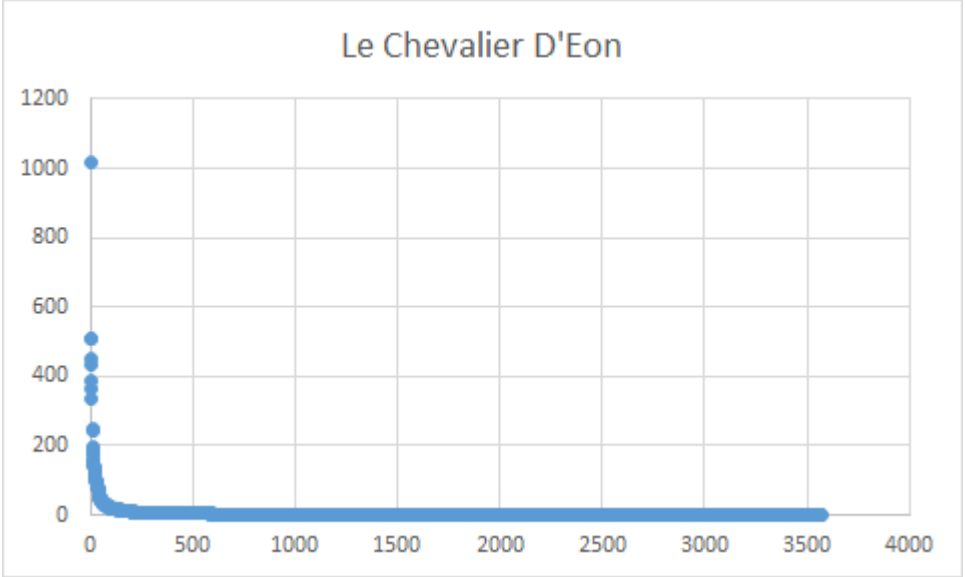
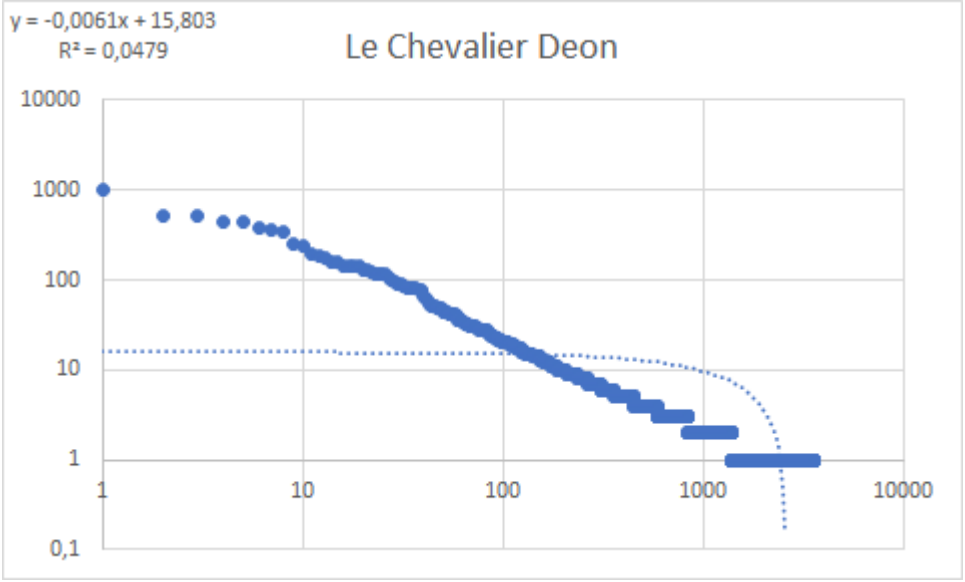


Gráfico log-log:



- Die Verwandlung
  Gráfico de ocurrencias:

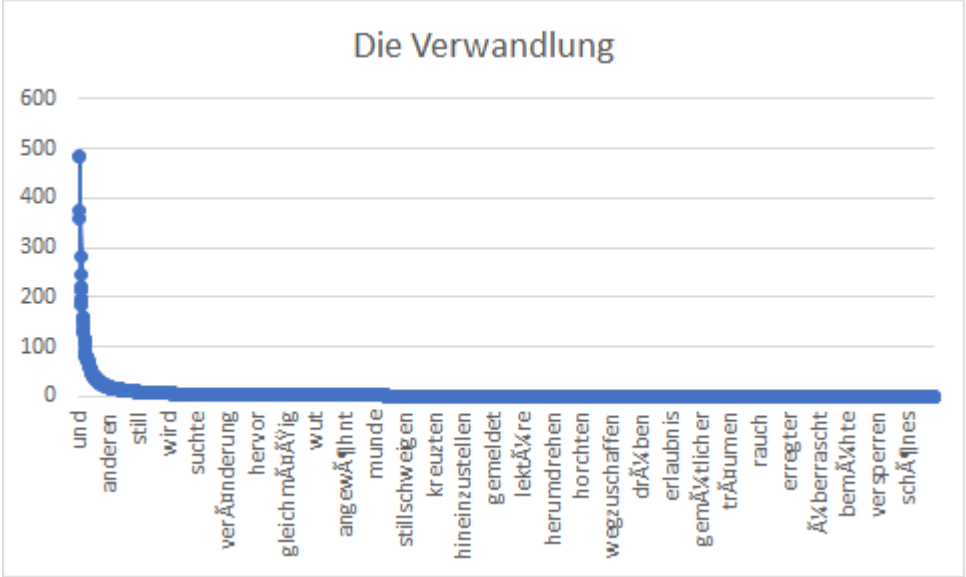
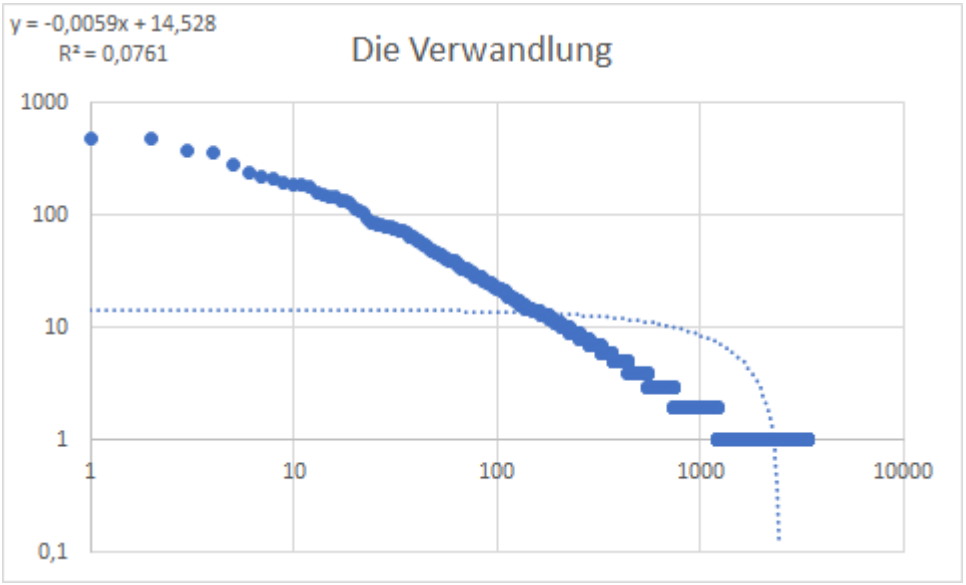


Gráfico log-log:



Vemos claramente que unas pocas palabras ocurren muy frecuentemente, un número medio de palabras tienen frecuencia media, y muchas palabras tienen una frecuencia muy baja. Vemos cómo los datos siguen una recta en escala logarítmica.

La mayor frecuencia claramente se ve que se lo llevan las palabras vacías, lo que nos muestra que cuanta mayor repetición menor interés, por eso se deben de extraer del conjunto de palabras. Esto aumenta claramente la eficacia de búsqueda.