

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**GRADO EN INGENIERÍA DEL SOFTWARE**

**Curso Académico 2019/2020**

**Trabajo Fin de Grado**

**Clasificación Automática de Logs de Testing de Software**

**Autor**: Daniel Diz Molinero

**Directores**: María del Soto Montalvo Herranz

Micael Gallego Carrillo

Resumen

**Los logs de testing son un componente muy importante para determinar el éxito de la ejecución de un programa. Proveen información clave para encontrar errores en el código. Estos errores se pueden agrupar en distintas categorías que representan el origen de los errores. Analizando cada categoría se puede extraer los puntos en común de los logs para automatizar el proceso.**

**En este Trabajo de Fin de Grado se diseña un clasificador de logs de testing que utilizando diversas técnicas y algoritmos compara la frecuencia de aparición de palabras de los distintos logs pertenecientes a distintas categorías de errores.**

**Se ha realizado con la librería de *Waikato Environment for Knowledge Analysis* (WEKA). En primer lugar, buscando y adaptando un dataset apropiado. Posteriormente, filtrándolo con el uso de algoritmos supervisados. Finalmente, utilizando la combinación de algoritmos de clasificación, se crea, entrena y evalúa un modelo capaz de predecir el tipo de error de un log.**

**Palabras clave: Minería de datos, Weka, algoritmos de clasificación, logs de testing, travis.**

Índice General

Índice de figuras 6

Índice de tablas 7

1. Introducción 9

2. Metodología y tecnologías 11

2.1. Metodología del software 11

2.2. Tecnologías empleadas 11

3. Descripción informática 14

3.1. Requisitos funcionales 14

3.2. Requisitos no funcionales 14

3.3. Casos de uso 15

3.4. Diagrama de clases 17

3.5. Planificación del proyecto 17

4. Sistema propuesto 20

4.1. Reordenado 20

4.2. Filtrado 20

4.3. Entrenamiento del modelo 22

4.4. Evaluación del modelo 24

5. Resultados 27

5.1. Dataset 27

5.1.1. Obtención del dataset 27

5.1.2. Adaptación del dataset 28

5.2. Medidas de evaluación 29

5.3. Resultados 32

6. Conclusiones y trabajo futuro 37

6.1. Trabajo futuro 37

Bibliografía 39

Anexos 42

Índice de figuras

Figura 2.1. Aplicación WEKA. Figura extraída de [1]. 12

Figura 3.1. Casos de uso del sistema. 15

Figura 3.2. Diagrama de clases. 17

Figura 3.3. Diagrama Grantt. 18

Figura 4.1. Arquitectura del sistema, fase FileReorderer. 20

Figura 4.2. Arquitectura del sistema, fase InstancesBuilder. 21

Figura 4.3. Arquitectura del sistema, fase Classification. 23

Figura 4.4. Arquitectura del sistema, fase Evaluation. 25

Figura 5.1. Figura de matriz de confusión simplificada a 2 clases. 30

Figura 5.2. Fórmula de F-Measure en la que el valor de Recall y Precision tienen el mismo peso. Figura obtenida de [9]. 30

Figura 5.3. Fórmula de la medida MCC para 2 clases. Figura obtenida de [10]. 31

Figura 5.4. Fórmula de la medida MCC para multi clases. Figura obtenida de [10]. 31

Figura Anexo.1. Estructura de los archivos del código del programa. 42

Índice de tablas

Tabla 3.1. Tabla de requisitos funcionales. 14

Tabla 3.2. Tabla de requisitos no funcionales. 14

Tabla 3.3. Descripción del caso de uso ‘generar archivos ARFF’. 16

Tabla 3.4. Descripción del caso de uso ‘generar modelo’. 16

Tabla 5.1. Comparativa entre los repositorios de la colección original y la modificada. 28

Tabla 5.2. Tabla de la matriz de confusión de la evaluación del modelo. 33

Tabla 5.3. Tabla de resultados de la evaluación del modelo. 34

Tabla 5.4. Tabla de resultados simplificados de la evaluación. 35

1. Introducción

Los logs de testing son artefactos decisivos para el desarrollo de un software libre de errores. Proveen de un resumen detallado de la ejecución de prueba del programa, indicando principalmente si ha tenido éxito o ha fallado. Un buen log, además, contiene información adicional, como los pasos seguidos, el origen de los fallos o las razones de las operaciones que han fracasado. Analizar la enorme cantidad de logs que se generan diariamente en tiempo real es un desafío imposible para ser realizado de forma manual.

Los motivos por los que un programa puede fallar son muchos, pero la mayoría de las veces se pueden trazar a un problema original. Los logs de testing son un tipo de texto de longitud muy variable, desde pocas decenas de líneas hasta los miles, pero el error original que causa que la ejecución falle completamente está escondido en unas pocas líneas. El trabajo manual de encontrar dicho error puede llevar al desarrollador una cantidad de tiempo considerable. A este problema hay que añadir el factor humano, ya que el error original genera varios errores más, el origen puede no ser encontrado adecuadamente.

Pasa solventar este problema se ha desarrollado este Trabajo de Fin de Grado, el cual se encarga de generar un modelo entrenado que se encarga de clasificar de forma automática los logs, con un porcentaje de acierto aceptable. La automatización proporciona una serie de beneficios que difícilmente se obtendrían de forma manual. Entre ellos se encuentra la consistencia, dado que el factor humano juega un gran papel en la clasificación manual, es difícil mantener un nivel de consistencia. Otros beneficios son el mayor contexto, la clasificación automática toma todo lo aprendido y lo aplica a cada log; la persistencia, una persona solo puede manejar un cierto número de logs; y el coste efectivo, un clasificador automático es capaz de manejar miles de logs mucho más rápido que una persona.

El resto de la memoria se estructura como sigue. En el capítulo 2 se describe la metodología y tecnologías utilizadas a la hora de desarrollar el proyecto. El capítulo 3 detalla la descripción informática. El capítulo 4 explica el sistema propuesto y sus diferentes fases. En el capítulo 5 se describe el dataset utilizado y los resultados obtenidos. Finalmente, el capítulo 6 presenta las conclusiones y trabajos futuros.

1. Metodología y tecnologías

En este capítulo se presentan la metodología de software y las tecnologías empleadas en este trabajo.

* 1. Metodología del software

La aplicación ha sido desarrollada utilizando un proceso de software iterativo, en el cual, en cada iteración, se ha ido perfeccionando el sistema hasta alcanzar el estado final.

La metodología utilizada es parecida a la metodología en espiral. Cada iteración consistía de diferentes fases que se repetían en las distintas iteraciones pero cada vez con mayor dificultad y peso. Investigación, diseño, codificación y evaluación. Todas estas fases se repetían en cada iteración con ligeros cambios según al problema asociado.

* 1. Tecnologías empleadas

El lenguaje de desarrollo elegido fue Java [1], por los conocimientos que se tenían sobre él y su popularidad, era el lenguaje ideal para este propósito. Java, además, permitió la selección de librerías entre una variedad de opciones.

La librería más importante era aquella que tendría las capacidades para filtrar, clasificar y evaluar el modelo que se generaría. De la numerosas a disposición, Weka [2] fue la seleccionada. Weka es una colección de algoritmos para la minería de datos. Además de contar con una API para su utilización en código, cuenta con una GUI con las mismas funcionalidades con la que se puede aprender y probar los diversos algoritmos, tal como se puede ver en la Figura 2.1.

Una captura de pantalla de una red social

Descripción generada automáticamente

Figura 2.1. Aplicación WEKA. Figura extraída de [1].

Una vez conocidos los mecanismos básicos de la librería, es fácil llevarlo al código. Weka es fácil de aprender, permite experimentar con varios clasificadores aplicándolos al mismo dataset, para encontrar aquel que mejores resultados obtenga.

Para finalizar, los diversos diagramas de este documento fueron realizados con las aplicaciones Draw.io [3] y Smartsheet [4].

1. Descripción informática

En este capítulo se presentan los requisitos del sistema, casos de uso, diagrama de clases y la planificación del trabajo realizado.

* 1. Requisitos funcionales

Un requisito funcional es un requerimiento de un sistema de software que define las operaciones y actividades que debe desempeñar. En la Tabla 3.1 se representan los requisitos funcionales de la aplicación desarrollada.

|  |  |
| --- | --- |
| RF1 | Se debe generar un modelo predictivo |
| RF2 | El programa debe ser capaz de clasificar una colección de logs en diferentes categorías |
| RF3 | Los datos deben ser filtrados para adecuarlos al clasificador |
| RF4 | Se necesitará una colección para probar el clasificador |
| RF5 | Se debe automatizar el proceso completo |
| RF6 | Se evaluará el clasificador para aproximarlo a su uso real |

Tabla 3.1. Tabla de requisitos funcionales.

* 1. Requisitos no funcionales

Un requisito no funcional es un requerimiento de un sistema de software que señala una restriccion del mismo. En la Tabla 3.2 se representan los requisitos no funcionales de la aplicación desarrollada.

|  |  |
| --- | --- |
| RNF1 | El índice de instancias clasificadas correctamente debe ser igual o superior al 80% |
| RNF2 | La clasificación debe tardar un tiempo inferior a 2h |
| RNF3 | La colección de prueba debe ser inferior a la de entrenamiento |
| RNF4 | El sistema debe funcionar independientemente del lenguaje de programación de los logs |
| RNF5 | Los datos deben ser equilibrados, extensos y categorizados |
| RNF6 | Informará de los pasos que ejecute |
| RNF7 | Utilizar formatos estándar para guardar el modelo y los datos |
| RNF8 | Todas las clases tendrán un mínimo del 60% de instancias clasificadas correctamente |

Tabla 3.2. Tabla de requisitos no funcionales.

* 1. Casos de uso

Las funcionalidades de la aplicación se agrupan en casos de uso, los cuales se muestran en el diagrama de casos de uso de la Figura 3.1. Como se puede observar, se trata de un número pequeño de casos de uso y de actores.

*Imagen que contiene texto

Descripción generada automáticamente*

Figura 3.1. Casos de uso del sistema.

Se trata de casos de uso enlazados, que se complementan y todos juntos forman un proceso. Por ejemplo, la Tabla 3.3, muestra la descripción del caso de uso ‘generar archivos ARFF’.

|  |  |
| --- | --- |
| Nombre | Generar archivos ARFF |
| Descripción | Adapta los logs a un formato entendible y divide en entrenamiento y pruebas |
| Flujo principal | 1. - El sistema busca los logs ordenados por categoría  2. - El sistema busca los archivos ARFF  3. - El sistema carga los archivos y los acomoda indicando el atributo que indica a que clase pertenece. |
| Flujo alternativo | En caso de no encontrar los logs reordenados:  1A. - El sistema ejecuta el caso de uso “Reordenar archivos”  En caso de no existir los archivos ARFF:  2A. - El sistema carga los logs ordenados  2B. - El sistema filtra los logs  2C. - El sistema divide los logs en entrenamiento (train) y pruebas (test)  2D. - El sistema genera y guarda los logs de entrenamiento y pruebas en archivos ARFF |

Tabla 3.3. Descripción del caso de uso ‘generar archivos ARFF’.

Otro ejemplo (ver Tabla 3.4) trata el caso de uso ‘generar modelo’.

|  |  |
| --- | --- |
| Nombre | Generar modelo |
| Descripción | Crea y entrena un modelo de predicción con los logs de entrenamiento |
| Flujo principal | 1. - El sistema crea un clasificador base  2. - El sistema ejecuta la técnica ‘boosting-bagging-stacking’  3. - El sistema vota los distintos clasificadores utilizados para encontrar la mejor combinación  4.- El sistema genera y guarda el modelo en un archivo |
| Flujo alternativo | En caso de existir un modelo:  1A. - El sistema carga el modelo en el programa |

Tabla 3.4. Descripción del caso de uso ‘generar modelo’.

* 1. Diagrama de clases

Captura de pantalla de computadora

Descripción generada automáticamenteLas diferentes clases de las que consta el proyecto se han agrupado en diferentes paquetes para juntar funcionalidades, tal y como se puede ver en la Figura 3.2. El paquete *sorter* agrupa las clases que se centran en reordenar los logs por categoría. Las clases dentro del paquete *text* se centran en el filtrado y clasificado de los logs. Finalmente, el paquete *util* se ocupa de las posibles utilidades que utilicen el resto de paquetes, entre ellas las rutas a los diferentes archivos.

Figura 3.2. Diagrama de clases.

* 1. Planificación del proyecto

La planificación del proyecto se divide en tres fases: conocer el problema, conocer la librería y clasificar el texto. En la Figura 3.3 se muestran las diferentes fases y como cada una de ellas complementa a la anterior y es necesario terminar una fase para avanzar a la siguiente.

Progresivamente las fases se iban dificultando y su duración aumentaba, aunque el conocimiento general aumentaba día a día, el tamaño del problema también, por lo que realizar cambios suponía esperar cada vez más tiempo para obtener resultados.

Empezando por conocer el problema, fue una fase de aprender y conocer nuevos conceptos, entender el problema y buscar soluciones. Uno de los problemas era qué librería utilizar, entre OpenNLP [5] y Weka. Finalmente, se decidió por Weka.

Una vez seleccionada la herramienta había que entenderla. Esta fase era complicada porque tratar con el problema de clasificar desde el principio requiere tener ya un dataset, y no se tenía. Se empezó por una *Graphical User Interface* (GUI) y se adaptó luego al lenguaje elegido, Java.

Finalmente, había que aproximarse más al problema y se empezó a clasificar texto real. Una vez obtenidos resultados satisfactorios había que buscar un dataset adecuado. Por último, se adaptó el clasificador al dataset.

Imagen que contiene captura de pantalla

Descripción generada automáticamente

Figura 3.3. Diagrama Grantt.

1. Sistema propuesto

El sistema completo se puede dividir en una serie de procesos con funcionalidades distintas, pero entrelazados. Cada proceso complementa al siguiente con una serie de información que será necesaria.

* 1. Reordenado

La primera fase consiste en transformar el dataset entero en un subconjunto de este con el que trabajar, siguiendo el esquema mostrado en la Figura 4.1.

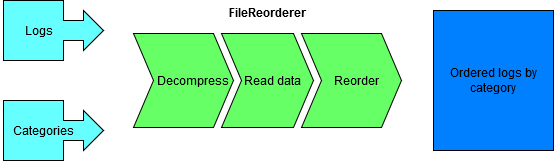


Figura 4.1. Arquitectura del sistema, fase FileReorderer.

Esta fase se describe mejor en el capítulo 5.1.3, en el que se describe la división del dataset y las decisiones tomadas. En líneas generales: se eliminan unas partes no apropiadas, se corrigen errores del propio dataset y finalmente se ordena por categoría, por tipo de error.

* 1. Filtrado

Esta fase se encarga de transformar el dataset a un formato compatible con la librería, siguiendo el esquema mostrado en la Figura 4.2.

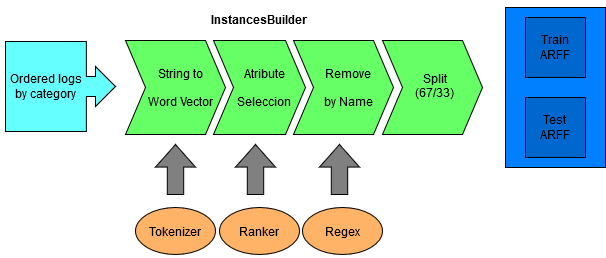


Figura 4.2. Arquitectura del sistema, fase InstancesBuilder.

El proceso de filtrado utiliza una serie de filtros de Weka para refinar el dataset. Empezando por transformar los logs a un formato reconocible. Con los logs separados por categorías en diferentes carpetas, utilizando la clase *TextDirectoryLoader* se es capaz de leer un directorio completo y convertirlo al formato que utiliza Weka para los dataset, la clase *Instances*.

Esta clase agrupa todo el dataset bajo un mismo objeto. Tras ser almacenado, cada log es guardado como un string entero, además de la clase a la que pertenece, un string con el nombre de la carpeta. Antes de proseguir, es necesario indicar la clase, esto es fácil ya que se trata del último atributo de las instancias.

Una vez preparado, se utiliza un filtro llamado *StringToWordVector* [5], que se encarga de transformar el texto de cada log en un vector de palabras con un peso distinto a cada una. Pero antes necesita ser capaz de separarlas, pues existen conjuntos de palabras que separados no dan la misma información que juntos. Para ello se utiliza un tokenizador, en concreto *NGramTokenizer*, que separa el texto en palabras delimitadas por varios caracteres, sea un espacio, una tabulación o un signo de puntuación entre varios. Después, evalúa y hace las combinaciones adecuadas de hasta un máximo de 3 palabras seguidas (por límites de memoria no se pudo subir este número).

Tras tener listo el tokenizador, se activan las opciones de Term Frequency (TF) y Inverse Document Frequency (IDF) [6]. Se utilizan para ponderar los valores del vector en función de la frecuencia con la que aparece un documento de un texto y el inverso de la frecuencia que aparece en todos los documentos.

Finalmente, el texto es filtrado y todos los logs se transforman en un vector de palabras. Esto permite más adelante usar los clasificadores, puesto que clasificar los logs enteros es tarea imposible.

El siguiente filtro usado es *AttributeSelection*. Se trata de un filtro que dado un vector de palabras ordena y selecciona las mejores palabras, aquellas cuya aparición en un texto lo acercan más a una categoría que a las otras. Para ello utiliza un ranking, *BestFirst*, y un evaluador, *CfsSubsetEval*. Este evaluador en concreto evalúa el valor de un subconjunto de atributos en función de su capacidad predictiva individual de cada categoría.

A continuación, utilizamos *RemoveByName*, un filtro que elimina todos aquellos atributos cuyo nombre no cumpla con una expresión concreta. El motivo principal de su uso es eliminar todos aquellos atributos que tengan caracteres no legibles, los cuales se pueden definir de muchas maneras, pero se ha definido de tal forma que solo acepte caracteres de UTF-8.

Finalmente, se debe realizar la separación entrenamiento y pruebas. Para ello el filtro *StratifiedRemoveFolds* es ideal. Este filtro divide las instancias completas de forma equilibrada para las clases, de tal forma que todas tienen el mismo porcentaje de instancias de entrenamiento y de pruebas. Se divide todo en tres partes y una para pruebas y el resto para entrenamiento, de tal modo que el 33,3% de los logs son para pruebas y el 66,7% para entrenamiento. Todo esto manteniendo un equilibrio entra clases.

* 1. Entrenamiento del modelo

Durante esta fase se utilizan una serie de algoritmos de clasificación para generar un modelo, siguiendo el esquema mostrado en la Figura 4.3.

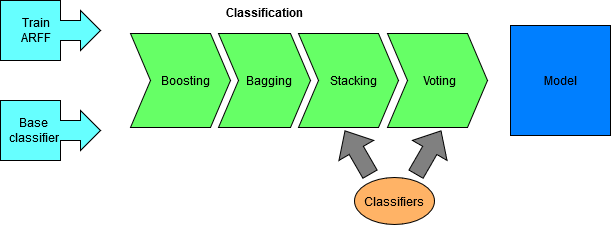


Figura 4.3. Arquitectura del sistema, fase Classification.

El entrenamiento del modelo en Weka utiliza varios algoritmos de clasificación para crear y entrenar un modelo en base a unos datos dados. Todo esto empieza seleccionando un clasificador base.

Weka agrupa numerosos clasificadores de muchos tipos, por lo que encontrar el mejor es una tarea difícil y que lleva tiempo. Tras varias decenas de pruebas se llegó a un muy buen clasificador *IBk* [7]. Este algoritmo no genera un modelo directamente, sino que genera una predicción para una instancia de pruebas creada en el momento. El algoritmo utiliza una función de distancia indicada, en este caso *LinearNNSearch*, para localizar las k instancias más cercanas en los datos de entrenamiento y utiliza esas seleccionadas para hacer una predicción.

Un algoritmo solo no es suficiente para obtener un modelo que cumpla con los requisitos. Hay que mezclar algoritmos para mejorar su precisión. Para ello existe una técnica Boosting-Bagging-Stacking que utiliza unos algoritmos que, aunque débiles ellos solos, al juntarlos se obtiene un clasificador con mejor rendimiento.

Esta serie de técnicas empieza por la llamada Boosting, en Weka con el algoritmo *AdaBoostM1*, es un método que comienza con un clasificador base, el *IBk* que hemos creado antes, y después utiliza un segundo clasificador para centrarse en las instancias que el primer clasificador se equivocó. El proceso continúa hasta un número límite de clasificadores o hasta una precisión indicada. Se seleccionaron 20 iteraciones y se probó hasta 100, pero la precisión no aumentaba pasadas las 20.

La segunda técnica, Bagging (igual en Weka), es un método que crea muestras separadas de todo el conjunto de datos y crea un clasificador para cada muestra. Los resultados de estos se combinan más adelante. El factor clave es que cada muestra sea diferente, dando a cada clasificador un enfoque y perspectiva ligeramente diferente frente al problema. Ayuda a reducir la varianza, y se basa directamente del modelo que genera el paso anterior.

Stacking, con mismo nombre en Weka, es una técnica donde se preparan múltiples algoritmos de antemano y un meta-algoritmo, en este caso directo del paso anterior. El meta-algoritmo aprende a tomar las predicciones de cada clasificador y a hacer predicciones precisas sobre datos no vistos.

Los algoritmos que seleccionemos son de mucha importancia para obtener los mejores resultados. Tras las pruebas realizadas previamente para obtener un clasificador base, se analizó que unos clasificadores mostraban unos mejores resultados que otros, entre ellos *J48*, *NaiveBayes* y *RandomTree*. Estos clasificadores junto a un nuevo *IBk* son los que se usan en este paso.

Finalmente, cada clasificador devuelve una serie de valores, y la combinación de todos estos es lo que se quiere, por ello se realiza una votación por peso utilizando *Voting* de Weka. El clasificador del paso anterior, junto a los seleccionados previamente son necesarios para la votación.

* 1. Evaluación del modelo

Finalmente, el modelo es evaluado y se obtienen una serie de resultados indicando el rendimiento de este, siguiendo el esquema mostrado en la Figura 4.4.

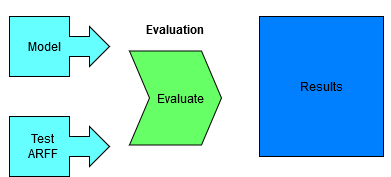


Figura 4.4. Arquitectura del sistema, fase Evaluation.

Existen varias formas de evaluar un modelo en Weka. El método *cross-validation* consiste en dividir los datos en dos grupos y emplear uno para entrenar y otro para pruebas. El problema con este método es que el modelo se ha entrenado con los datos de pruebas, por lo que no se trata de una predicción ciega. Por ello utilizaremos los datos de prueba que hemos separado previamente.

Estos datos han sido filtrados por los mismos filtros por lo que son completamente compatibles con los de entrenamiento. La evaluación simula un entorno real en el que el modelo no ha sido entrenado con los datos a probar. Una vez evaluado, hay que analizar los resultados obtenidos.

1. Resultados

En esta sección se detallará sobre el dataset seleccionado y su adaptación, las medidas de evaluación y los resultados obtenidos.

* 1. Dataset

En esta sección se especificará sobre el dataset seleccionado y su adaptación.

* + 1. Obtención del dataset

Uno de los aspectos más importantes para la realización de este proyecto es encontrar un dataset extenso, equilibrado y perfectamente categorizado. Con ello, se podrán extraer un número adecuado de instancias de cada clase y tener una clasificación lo más estable para todas las categorías. A continuación, se expondrán los tratamientos que el conjunto ha recibido y cómo se ha adecuado a los requisitos del proyecto.

Encontrar un dataset tan específico sin tener referencias puede llegar a ser una tarea ardua. Google es una herramienta poderosa para encontrar información, para ello desarrollaron una plataforma conocida como “Dataset Search”. Esta herramienta indexa millones de datos y con palabras clave es fácil encontrar lo que se busca. Una simple búsqueda usando la palabra “Travis” [7], una popular herramienta para la integración continua (CI) llevó a encontrar un conjunto que cumplía con los requisitos [8].

El dataset se compone de los repositorios Git de los que se extrajeron los logs de Travis, una serie de archivos JSON (JavaScript Object Notation) que categorizan cada log a una categoría, además de indicarle unas palabras clave, un archivo CSV (Comma Separated Values) que mapea el identificador de Travis a cada repositorio Git y, finalmente, para cada repositorio los logs de un “job” de una “build” de Travis (cada “build” puede tener múltiples “jobs”).

El dataset se compone de varias decenas de miles de logs de 16 repositorios diferentes. Java es el lenguaje principal de todos ellos, esto se trata de una coincidencia y no era uno de los objetivos deseados, se buscaron otros conjuntos con una mayor variedad de lenguajes, pero no se encontraron conjuntos que cumplieran los requisitos tan excelentemente.

* + 1. Adaptación del dataset

Una vez obtenidos todos los datos necesarios hay que realizar una serie de pasos para adaptarlo a nuestro problema. Empezando por los pasos manuales, el número de archivos JSON que tenemos no se corresponde con el número de repositorios, como se muestra en la Tabla 5.1, por ello hay que descartar aquellos repositorios que no tengan su correspondiente JSON.

|  |  |
| --- | --- |
| *Colección original* | *Colección final* |
| 1129565 | 1129565 |
| 132351 | 132351 |
| 1501298 | 1501298 |
| 1504827 | 1504827 |
| 1512319 | 1512319 |
| 1889385 |  |
| 2026694 | 2026694 |
| 2036767 | 2036767 |
| 2136677 |  |
| 2853863 | 2853863 |
| 2964082 | 2964082 |
| 297539 |  |
| 380237 |  |
| 407298 | 407298 |
| 487759 |  |
| 78264 | 78264 |

Tabla 5.1. Comparativa entre los repositorios de la colección original y la modificada.

El segundo cambio manual por realizar se trata de un error por parte del creador de la colección. Los archivos JSON funcionan como tablas clave-valor, para cada clave única existe un documento valor. El archivo denominado como “logcat.1129565.json” tiene un error por el cual tiene dos claves idénticas. Por ello, es necesario cambiar una de ellas o combinar la información. Revisando ambas, la primera aparición, empezando por la parte superior del documento, contiene tan solo un par de logs, mientras que la segunda aparición tiene considerablemente muchos más. Tomados estos factores en cuenta se decidió que la segunda aparición de la clave se llamaría de forma distinta y sería una categoría nueva que clasificar. El nombre de la nueva clave sería el mismo que el original, pero con un ‘2’ después.

Una vez resueltos estos problemas, la colección esta lista para trabajar con ella, empezando por una ordenación por categorías.

Los logs están originalmente ordenados por repositorio, pero esto no nos sirve. Empezamos creado una tabla con índice el nombre del log y de contenido el log en sí. Para obtener la información de a qué categoría pertenece miramos los JSON. Por cada uno, miramos cada clave-valor, y creamos una tabla nueva con clave la categoría y valor una lista de los logs. Aquí es donde ocurre el cambio más importante.

Dados los requisitos en la búsqueda de la colección, se hace notar el gran tamaño de esta. Tanto es así que si no lo limitásemos empezarían a saltar problemas de memoria en la clasificación. Con el método de prueba y error, se llegó a 70 logs por categoría como máximo, si bien es cierto que hay alguna categoría que queda ligeramente desbalanceada, la gran mayoría lo están, y reducir la selección podría causar incertidumbre sobre la fiabilidad del modelo. Por estos motivos, 70 logs por categoría son añadidos a nuestra tabla de categorías. Estos logs no se seleccionan de una forma definida, pero tampoco es de forma aleatoria. El orden de lectura es el que selecciona estos logs.

Finalmente, ya con la tabla de categorías y logs creada, leemos cada categoría y le creamos un directorio nuevo para añadir sus logs correspondientes. Para cada log de la lista, comprobamos que exista físicamente al archivo con la primera tabla que creamos. Una vez comprobado copiamos la información con una nueva ruta y creamos el archivo. Una vez reordenado todo se puede proceder al siguiente paso, la clasificación de los logs.

* 1. Medidas de evaluación

La matriz de confusión (ver Figura 5.1) se trata de un concepto muy útil para ver el desempeño de nuestro modelo. Se trata de una matriz de nuestro problema de tamaño NxN, siendo N el número de clases. Sirve para mostrar cuando una clase es confundida por otra.

Captura de pantalla de un celular

Descripción generada automáticamente

Figura 5.1. Figura de matriz de confusión simplificada a 2 clases.

Esta matriz nos proporciona una serie de medidas que se utilizarán para calcular otras más complejas más adelante. Para ello pongamos un ejemplo, grupo A y grupo B:

* **TP**: Verdaderos positivos, si el modelo dice A y la realidad es A
* **FN**: Falso negativos, si el modelo dice B y la realidad es A
* **FP**: Falsos positivos, si el modelo dice A y la realidad es B
* **TN**: Verdaderos negativos, si el modelo dice B y la realidad es B

En base a estos valores se calculan otros más avanzados que nos dan más información:

* **TP Rate** (True Positives rate): La ratio de verdaderos positivos, instancias correctamente clasificadas con la clase dada. También llamada sensibilidad.
* **FP Rate** (False Positive rate): La ratio de falsos positivos, instancias clasificadas incorrectamente según la clase dada. También llamada especificidad.
* **Precision**: proporción de instancias que son verdaderamente de una clase entre el total de instancias de esa clase.
* **Recall**: proporción de instancias clasificadas como una clase dada entre el total real, equivale a la tasa de la ratio de verdaderos positivos (TP rate).
* **F-Measure**: Una medida que combina precisión y recall. La Figura 5.2 muestra como es calcula.

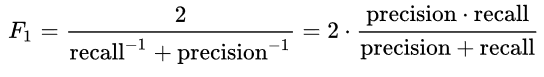


Figura 5.2. Fórmula de F-Measure en la que el valor de Recall y Precision tienen el mismo peso. Figura obtenida de [9].

* **MCC** (Matthew Correlation Coefficient): Una medida que calcula la calidad de una clasificación binaria entre clases [9]. La Figura 5.3 muestra la forma de calcularla.



Figura 5.3. Fórmula de la medida MCC para 2 clases. Figura obtenida de [10].

Para más clases la fórmula, mostrada en la Figura 5.4, se complica un poco.

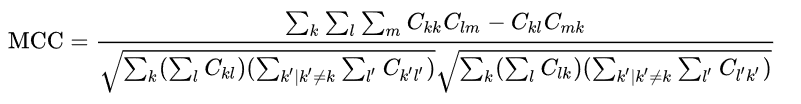


Figura 5.4. Fórmula de la medida MCC para multi clases. Figura obtenida de [10].

* **ROC Area** (Receiver Operating Characteristic): Se trata de una de las medidas más populares en esta industria. Se le conoce también como la ratio de falsos positivos y de verdaderos positivos
* **PRC Area** (Precision-Recall Curve): Se trata de una medida muy efectiva cuando se trata con clasificaciones binarias, especialmente con datasets desequilibrados [10].

También se proporcionan unos resultados simplificados:

* **Correctly Classified Instances:** Número y porcentaje de instancias clasificadas correctamente.
* **Incorrectly Classified Instances:** Número y porcentaje de instancias clasificadas incorrectamente.
* **Kappa statistic:** Una medida para ajustar el efecto del azar.
* **Mean absolute error**: Sirve para cuantificar la precisión de una predicción comparando los datos de entrenamiento con los de pruebas.
* **Root mean squared error**: Conocida como desviación típica de los errores cuando se hace una predicción a un dataset.
* **Relative absolute error**: Se utiliza para medir el rendimiento de un modelo predictivo en base a una clasificación simple. En Weka, el clasificador simple se llama *ZeroR*.
* **Root relative squared error**: Se trata de una medida relativa de lo que un clasificador simple habría obtenido.

Weka nos calcula y proporciona todas estas medidas, pero no todas son útiles para nuestro caso. ROC area es una medida que indica que tan bueno puede un modelo distinguir entre dos clases. ROC area es mejor que F-Measure cuando no se tienen clases pequeñas [12].

* 1. Resultados

Weka muestra los resultados de tres formas para poder analizarlos hasta el más mínimo detalle.

La matriz de confusión de nuestro modelo es más complicada (ver Tabla 5.2), puesto que hay un gran número de clases. Pero tras obtener los resultados vemos que la mayoría está a cero. Se han marcado de un azul más oscuro, en la Tabla 5.2*,* todos los verdaderos positivos, aunque se pueden encontrar fácilmente siguiendo la diagonal de la matriz. También se han marcado, en rojo, todas aquellas casillas que no están a cero. Una primera observación indica que en la mayoría de las predicciones el modelo ha acertado.

Mirando clase por clase, se puede ver que la primera clase, *androidsdk*, se encuentra vacía, con cero instancias, tanto en los datos de prueba como en los de entrenamiento. Esto indica un error en el dataset original que no afecta a los resultados. El dataset contiene en algún JSON alguna entrada, pero sin documentos enlazados. Otra categoría por observar es *lint*, esta contiene un número minúsculo de instancias y la hace potencialmente descartable. Finalmente, la categoría *license* tiene un número reducido y sería prudente equilibrarlo con el resto.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** |  |  | **classified as** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **a** | **androidsdk** |
| 0 | 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **b** | **buildconfig** |
| 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **c** | **buildconfig2** |
| 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **d** | **buildout** |
| 0 | 2 | 0 | 0 | 20 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **e** | **checkstyle** |
| 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **f** | **compile** |
| 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **g** | **compile\_clj** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **h** | **crash** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **i** | **dependency** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **j** | **findbugs** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **k** | **git** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **l** | **incompatibility** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **m** | **itestfailure** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **n** | **javadoc** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |  | **o** | **license** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  | **p** | **lint** |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 0 | 0 | 0 | 0 |  | **q** | **other** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 0 |  | **r** | **passed** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 | 0 | 0 | 0 | 0 |  | **s** | **pmd** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23 | 0 | 0 | 0 |  | **t** | **rat** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 22 | 0 | 0 |  | **u** | **sphinx** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 | 0 |  | **v** | **testfailure** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 23 |  | **w** | **testfailure\_clj** |

Tabla 5.2. Tabla de la matriz de confusión de la evaluación del modelo.

Con estos valores, de la tabla 5.2, Weka calcula los valores de las medidas TP Rate, FP Rate, Precision, Recall, F-Measure, MCC, ROC Area y PRC Area, que nos dan más información de cada clase.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TP Rate** | **FP Rate** | **Precision** | **Recall** | **F-Measure** | **MCC** | **ROC Area** | | **PRC Area** | **Class** |
| ? | 0 | ? | ? | ? | ? | | ? | ? | androidsdk |
| 1 | 0,004 | 0,923 | 1 | 0,960 | 0,959 | | 1 | 1 | buildconfig |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | buildconfig2 |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | buildout |
| 0,833 | 0,002 | 0,952 | 0,833 | 0,889 | 0,886 | | 0,966 | 0,922 | checkstyle |
| 1 | 0,004 | 0,920 | 1 | 0,958 | 0,957 | | 0,999 | 0,989 | compile |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | compile\_clj |
| 0,875 | 0 | 1 | 0,875 | 0,933 | 0,932 | | 0,997 | 0,945 | crash |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | dependency |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | findbugs |
| 1 | 0,011 | 0,821 | 1 | 0,902 | 0,901 | | 0,996 | 0,872 | git |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | incompatibility |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | itestfailure |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | javadoc |
| 0,933 | 0 | 1 | 0,933 | 0,966 | 0,965 | | 0,999 | 0,981 | license |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | lint |
| 0,875 | 0 | 1 | 0,875 | 0,933 | 0,932 | | 0,996 | 0,953 | other |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | passed |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | pmd |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | rat |
| 1 | 0,002 | 0,957 | 1 | 0,978 | 0,977 | | 1 | 0,996 | sphinx |
| 1 | 0 | 1 | 1 | 1 | 1 | | 1 | 1 | testfailure |
| 1 | 0 | 0,958 | 1 | 0,979 | 1 | | 1 | 1 | testfailure\_clj |
| 0,977 | 0,001 | 0,979 | 0,977 | 0,977 | 0,977 | | 0,998 | 0,984 | Weighted Avg. |

Tabla 5.3. Tabla de resultados de la evaluación del modelo.

Los valores de estas medidas (ver Tabla 5.3) estan normalizados, todos estan entre 0 y 1. En la mayoría de caso, salvo en la medida FP Rate, un valor de 1 es óptimo, mientras que 0 es lo peor. En el caso de FP Rate es lo opuesto, siendo 0 lo ideal y 1 lo peor.

De todas las medidas que Weka nos proporciona las más interesantes son ROC area y F-Measure. Puesto que la mayoría de las clases consta de al menos 20 casos, es mejor observar ROC area [13]. Todos los valores de ROC área estan por encima del 0.95 y la enorme mayoría, a excepción de un caso, se encuentra por encima del 0.99. Dados estos resultados queda claro que la clasificación ha sido un éxito [14].

Observando los resultados simplificados (ver Tabla 5.4) confirmamos el éxito del modelo. La medida correctly classified instances nos indica que más de un 97% de los logs se han clasificado en la clase correcta. Aún más, la medida root mean squared error (RMSE) se encuentra ligeramente por encima del 0.05. Este valor indica el error de los datos predichos frente a los del modelo, los valores más bajos indican un mejor ajuste.

|  |  |  |
| --- | --- | --- |
| Correctly Classified Instances | 469 | 97,7083% |
| Incorrectly Classified Instances | 11 | 2,2917% |
| Kappa statistic |  | 0,9759 |
| Mean absolute error |  | 0,0077 |
| Root mean squared error |  | 0,0508 |
| Relative absolute error |  | 9,3123% |
| Root relative squared error |  | 24,9458% |
| Total Number of Instances | 480 |  |

Tabla 5.4. Tabla de resultados simplificados de la evaluación.

1. Conclusiones y trabajo futuro

Se escogieron los campos del machine learning y de software testing al tratarse de campos que en el mundo del desarrollo software que están creciendo constantemente. Los logs de testing tienen diferentes usos, ya sea ejecuciones de prueba o construcciones de proyectos en integración continua. Es importante saber si un test ha fallado o no, y el porqué del fallo. Esto como tarea manual lleva un tiempo que la automatización puede resolver.

Durante el trabajo se han ido aprendiendo las diferentes herramientas y conceptos. En un principio se utilizaban datos ya filtrados y sencillos con los que jugar. Luego se cambió a una aproximación más real que necesitaba preprocesar antes de poder clasificar.

Cuando se obtuvo el dataset final se observó que el preprocesado realizado era insuficiente y era necesario realizar más. Otros cambios fueron los de hacer un clasificador mezclando diferentes técnicas, que mejoraban el modelo final.

* 1. Trabajo futuro

Tal como se indicó en su apartado, no se entrena el modelo con la totalidad de la colección por los diversos problemas de memoria, pero sería de interés incluir una mayor parte.

También un fallo de la colección es que son solo logs de Java, expandirlo a otros lenguajes y expandir a más errores más específicos del lenguaje permitiría ser usado por muchos más proyectos.

Por otra parte, existen una serie de frameworks que permiten la utilización de Weka para sistemas distribuidos, sean estos Spark [15] y Hadoop [16], que se podría utilizar Se trata de un plugin que permite ejecutar los diferentes algoritmos de Weka en clusters de máquinas. La motivación principal detrás de estos frameworks es tener un dataset demasiado grande que no quepa en la memoria RAM de un ordenador corriente. También si el tiempo de procesado es demasiado grande.

Bibliografía

[1] Java. <https://www.java.com/es>/ [Último acceso es 07/09/2020]

[2] Weka. [https://www.cs.waikato.ac.nz/ml/weka/](https://www.cs.waikato.ac.nz/ml/weka/%20) [Último acceso es 07/09/2020]

[3] Draw.io. [https://app.diagrams.net](https://app.diagrams.net/)/ [Último acceso es 17/08/2020]

[4] Smartsheet. <https://es.smartsheet.com/> [Último acceso es 17/08/2020]

[5] OpenNLP. <https://opennlp.apache.org/> [Último acceso es 07/09/2020]

[6] QAISER, Shahzad; ALI, Ramsha. Text mining: use of TF-IDF to examine the relevance of words to documents. International Journal of Computer Applications, 2018, vol. 181, no 1, p. 25-29.

[7] Travis CI. <https://travis-ci.org/> [Último acceso es 07/09/2020]

[8] Rausch, Thomas. (2018). Java OSS Travis-CI Build Failure Dataset [Data set]. Zenodo. <http://doi.org/10.5281/zenodo.1745638> [Último acceso es 07/09/2020]

[9] Valor-F. Wikipedia. <https://es.wikipedia.org/wiki/Valor-F> [Último acceso es 17/08/2020]

[10] Coeficiente de Correlación de Matthew. Wikipedia. <https://en.wikipedia.org/wiki/Matthews_correlation_coefficient> [Último acceso es 17/08/2020]

[11] Saito, Takaya, and Marc Rehmsmeier. “The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets.” PloS one vol. 10,3 e0118432. 4 Mar. 2015, doi:10.1371/journal.pone.0118432.

[12] Elección de la mejor métrica para problemas de clasificación multi clase. <https://medium.com/usf-msds/choosing-the-right-metric-for-evaluating-machine-learning-models-part-2-86d5649a5428> [Último acceso es 10/09/2020]

[13] Hand, D.J., Till, R.J. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. Machine Learning 45, 171–186 (2001). <https://doi.org/10.1023/A:1010920819831>

[14]Rice, Marnie & Harris, Grant. (2005). Rice ME, Harris GTComparing effect sizes in follow-up studies: ROC Area, Cohen's d, and r. Law Hum Behav 29: 615-620. Law and human behavior. 29. 615-20. 10.1007/s10979-005-6832-7.

[15] Koliopoulos, Aris-Kyriakos & Yiapanis, Paraskevas & Tekiner, Firat & Nenadic, Goran & Keane, John. (2015). A Parallel Distributed Weka Framework for Big Data Mining Using Spark. 10.1109/BigDataCongress.2015.12.

[16] Hadoop. <https://hadoop.apache.org/> [Último acceso es 07/09/2020]

Anexos

A continuación, se detallará el proceso para ejecutar el código que nos permite generar nuestro modelo. En primer lugar, el sistema consta de dos ejecutables: la clase *FileReorderer* y la clase *ModelBuilder* que se muestran en la Figura Anexo.1.

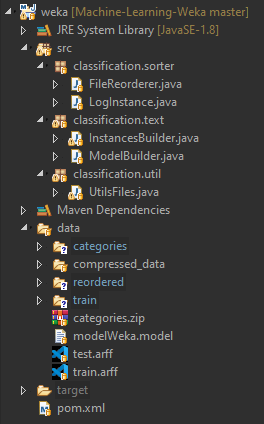


Figura Anexo.1. Estructura de los archivos del código del programa.

*FileReorderer* utiliza el dataset original, tras los cambios manuales realizados, y reordena y equilibra los logs por tipo de error. Para ello requiere en su ruta indicada, en la carpeta ‘data’ de la aplicación, los logs y la información de sus categorías.

Para reordenar los logs por categorías es necesario tener el archivo ‘categories.zip’ que contiene la información de cada repositorio con cada log de forma comprimida y la carpeta ‘compressed\_data’, que contiene una subcarpeta por cada repositorio, que, a su vez contiene cada log de forma comprimida. Al ejecutar la clase *FileReorderer* el sistema descomprimirá las categorías en la carpeta ‘categories’ y los logs comprimidos en la carpeta ‘train’. Una vez descomprimido todo, se encarga de ordenarlo en la carpeta ‘reordered’ por categoría.

*ModelBuilder* se encarga de preprocesar los logs y generar el modelo. Al ejecutarla, empezara buscando si existe la carpeta ‘reordered’. En caso de no existir, ejecutara el anterior ejecutable, *FileReorderer*. Después, busca los archivos ‘train.arff’ y ‘test.arff’ que contiene la información de los logs filtrada y dividida para su uso posterior. En caso de no existir estos archivos, procede a cargar los logs directamente de la carpeta ‘reordered’ y los filtra, para finalmente guardarlos en ‘train.arff’ y ‘test.arff’. A continuación, busca el archivo ‘modelWeka.model’ que contiene el modelo ya entrenado. En caso de no existir, utilizando los logs filtrados en el paso anterior, ‘train.arff’, lo genera. Finalmente, con el modelo y los logs de prueba filtrados, ‘test.arff’, evalúa el modelo.