[49]

Mi, Q., Keung, J., Xiao, Y., Mensah, S., & Gao, Y. (2018). Improving code readability classification using convolutional neural networks. Information and Software Technology, 104, 60–71. Retrieved from <https://linkinghub.elsevier.com/retrieve/pii/S0950584918301496>

**KEYWORDS AUTOR:**

Code Readability, Convolutional Neural Network, Deep Learning, Program Comprehension, Empirical Software Engineering, Open Source Software

**TÍTULO**:

Mejora de la clasificación de legibilidad de código utilizando redes neuronales convolucionales

**PARA ANEXAR A DOCUMENTO:**

**\*\* INICIO \*\*\***

**DATASET**:

25000 fragmentos de proyectos Java de código abierto.

<https://github.com/CityU-QingMi/DeepCRM/tree/master/Code%20Snippet%20Repository>

**HERRAMIENTA**:

<https://github.com/CityU-QingMi/DeepCRM>

**LENGUAJE DE PROGRAMACIÓN:**

Java

**TAGS**

LEGIBILIDAD; MODELO; RED NEURONAL CONVOLUCIONAL; APRENDIZAJE PROFUNDO; EVOLUCION; DeppCRM; ConvNets;

**Nota**: Ver todo el resumen, este documento tiene aportes muy importantes para la legibilidad

**\*\* FIN \*\*\***

**RESUMEN COMPLETO**

Mi et al (2018) propusieron un modelo basado en las redes neuronales convolucionales, denominado ConvNets, que presentados en 3 diferentes granularidades, forman un framework de aprendizaje profundo, denominado DeppCRM, el cual permite la clasificación de la legibilidad del código fuente gracias al aprendizaje automático de características complicadas desde el código fuente.

El código fuente fue convertido en matrices de enteros para poder ser usado como entrada en el ConvNets. El nuevo modelo logra un 83.8% de precisión y 83.5% de medida f.

Las redes neuronales convolucionales son tipos especiales de redes neuronales con estructuras inspiradas en la biología. Se compone de las capas: convolucionales, pooling y totalmente conectada.

**EVOLUCIÓN DE LA LEGIBILIDAD**

|  |  |
| --- | --- |
| **Autor (año)** | **Aporte** |
| Binkley et al ([10] 2009) | Estudio para analizar los efectos de las convenciones para nombrar un identificador. |
| Sasaki et al [12] 2013 | Propone una técnica de re ordenamiento para acortar la distancia de una variable hasta su referencia. |
| Lee et al [13] 2013 | Probó que las violaciones en las convenciones de codificación afectan la legibilidad del código. |
| Wang et al [14] 2014 | Propuso un algoritmo para insertar líneas blancas automáticamente para separar los bloques significativos, mejorando la legibilidad. |
| Aggarwal et al [15] 2002 | Estimó la legibilidad por el promedio de todas la líneas para comentar líneas. |
| Borstler et al [17] 2016 | Propone el SRES, un puntaje fácil de legibilidad de software, para evaluar la legibilidad de acuerdo a la longitud promedio del lexema y al número promedio de palabras por declaración o bloque. |
| Buse et al [5] 2010 | Construye el primer modelo de legibilidad.  Precisión 76.5% |
| Dorn et al [8] | Propone un nuevo modelo de legibilidad, adicionando características visuales, espaciales y lingüísticas.  Precisión 78.6% |
| Posnett et al [7] 2011 | Presentó una teoría simple e intuitiva, basada en el tamaño y la entropía del código.  Precisión 71.5% |
| Scalabrino et al [9] 2016 | Propuso un conjunto de características textuales basadas en el análisis léxico del código fuente.  Defectos:  Inefectivo:  Precisión 73.9% |

**Deficiencias** de los modelos anteriores:

* Inefectivo: capturar información parcial sobre el código fuente, que puede no ser adecuada. Las características que se pueden usar para un conjunto de datos pueden no serlo para otro; y pueden existir redundancias y superposiciones entre las características diseñadas manualmente, lo que probablemente limitará el rendimiento del modelo.
* Ineficiente: La elaboración de buenas características es difícil y lleva mucho tiempo porque requiere un conocimiento específico de dominios sólidos. Además, los investigadores y los profesionales deben examinar la validez de todas las características prometedoras por separado o en combinación, lo que hace que el enfoque manual sea costoso o incluso imposible

**MODELO PROPUESTO**

El problema de clasificación de la legibilidad del código fuente, es abordado como un problema de clasificación binaria, donde dado un fragmento de código x, representado como una matriz mxn, donde m denota las líneas de código y n denota la longitud máxima de línea, es clasificado como legible o no legible.

Descripción del proceso:

1. Transformación del código fuente a una matriz de enteros, que será la entrada para las ConvNets. La representación es realizada en 3 diferentes tipos de granularidad:
   1. Representación a nivel de caracter.
   2. Representación a nivel de token (como palabras clave y operadores). Análisi léxico.
   3. Representación a nivel de nodo (como declaraciones y definiciones). Árbol de sintaxis abstracta.
2. Construcción de múltiples ConvNets, uno para cada representación, y entrenamiento con respecto a los parámetros que se pueden aprender. Las tres redes neuronales son ensambladas como una sola, se le ajustan los pesos y forman el modelo denominado DeepCRM. Cada ConvNet fue implementado en Python usando TensorFlow, y entrenado de manera supervisada usando el algoritmo Adam.
3. Se determina si una nueva instancia es legible o ilegible utilizando el modelo entrenado.

**DATASET**

Fragmentos de Buse, de Dorn, de Scalabrino y varios fragmentos de proyectos de código abierto obtenidos de GitHub y de SourceForge.

Métricas de evaluación: exactitud y la medida-f

**RESULTADOS**

* DeepCRM proporciona el mejor rendimiento, con una precisión del 83.8% y una medida f del 83.5%, lo que valida la efectividad del método de redes ensambladas.
* Mejora un 2.4% con respecto al modelo general de legibilidad que reúne todas las características de todos los modelos anteriores y 17.2% con respecto al modelo de Posnett.
* Para mitigar el riesgo del sobre entrenamiento, fueron recolectados 25000 fragmentos de código de proyectos de código abierto. Luego, se realizaron las pruebas de clasificación utilizando los conjuntos de datos de los modelos anteriores.

**RETOS**

* Ampliar el conjunto de datos utilizados a no sólo proyecto de código abierto, sino comerciales.
* Ampliar la diversificación de lenguajes de programación en los fragmentos de código, no sólo Java.
* Aplicar el modelo a una situación del mundo real, como integrarlo a un IDE o a un sistema de control de versiones para que los desarrolladores puedan monitorear de forma contínua la legibilidad del código fuente en sus proyectos.