Identificación de Elementos Web Faltantes Utilizando Técnicas de Procesamiento del Lenguaje Natural

Daniel Artavia and Gilbert Márquez
Universidad de Costa Rica
San José, Costa Rica

I. INTRODUCCIÓN

Los localizadores que cambian son una de las causas del fenómeno conocido como "flaky test" o prueba inestable. Estas son pruebas de software que en algunas ocasiones pasan y en otras no, aunque el funcionamiento del programa sea adecuado. La presencia de pruebas inestables implica costos en recurso humano para revisar y corregir los errores, e incluso pueden significar el tener que hacer las pruebas manualmente. Asimismo, los falsos positivos bloquean el pipeline del desarrollo con Integración Continua, retrasando el lanzamiento de nuevas funcionalidades y el proceso de otros equipos de desarrollo.

Cuando hablamos de un elemento faltante, nos referimos a un elemento que no puede ser encontrado por las pruebas debido al cambio de su localizador. En el momento en que una prueba falla debido a un elemento faltante, el desarrollador debe examinar el archivo HTML para encontrar el elemento causante del error, para ello, puede incluso ser necesario comparar con una versión anterior del código.

En este artículo pretendemos investigar si es posible utilizar el contexto en el DOM (Document Object Model) de un elemento HTML para identificar nuevas versiones de elementos cuyo identificador cambió. De modo que sea posible agilizar este proceso de corrección de errores causados por elementos faltantes, por lo tanto, se reducirían costos tanto de recurso humano como de tiempo durante el desarrollo de software.

En la actualidad se cuenta con algunas herramientas que intentan solucionar este problema. La principal herramienta de código abierto es Healenium, que funciona utilizando algoritmos basados en la detección de subcadenas más largas inicialmente. Este método de detección de cambios en componentes falla bajo algunos escenarios como cuando, aparte del elemento buscado, ocurren cambios en elementos vecinos [2]. Otro tipo de herramienta es el uso de plataformas privadas como Functionize, cuya funcionalidad se basa en el uso de visión de computadoras y usar múltiples localizadores por elemento. Esta forma de afrontar el problema es efectiva pero el proceso es complejo en términos de espacio requerido por prueba, y podría ser más eficiente.

Para lo anterior, proponemos el uso de técnicas de Procesamiento del Lenguaje Natural. Mediante un análisis de contexto, definido por la comparación de contenido y atributos de elementos HTML, buscamos determinar el elemento cuyo localizador ha sido modificado, así como al localizador que se modificó y desencadenó el error. En resumen, el programa recibe como entrada dos versiones de un archivo HTML, la versión anterior (funcional) y la versión actual (causante del error), y como salida se espera una posible solución al error, es decir, el localizador necesario para el elemento faltante.

II. TRABAJO RELACIONADO

Existen distintos trabajos relacionados a similitud entre elementos HTML. En particular, [1] propone una función para comparar el contenido de 2 listas HTML utilizando el contexto y [2] el cual compara distintos algoritmos de similitud y propone uno de mayor efectividad. Además, como se indica en [1], existen distintos trabajos relacionados a la similitud entre tablas HTML y la extracción de sus datos.

Sobre la solución de errores causados por cambios en localizadores, de momento, hemos encontrado las herramientas mencionadas anteriormente (Healenium y Functionize), las cuales, como se comentó, presentan ciertas limitaciones que pretendemos cubrir para lograr una corrección más efectiva.

Por otra parte, con respecto al uso de Procesamiento del Lenguaje Natural en archivos HTML, no hemos encontrado información, por lo que nuestra propuesta podría significar un aporte interesante.

III. METODOLOGÍA

Nuestro producto consta de 3 pasos: Tokenización, entrenamiento del modelo y construcción del archivo corregido.
Para la tokenización se utilizó BeautifulSoup, debido a que es la biblioteca más popular para Web Scraping de Python y sus funciones son muy simples. Este proceso consiste en extraer todas las etiquetas y los atributos que nos interesan (clases e identificadores) de un archivo HTML.
Primero, realizamos la tokenización del archivo original (el que funciona), como resultado obtendremos los Tokens que conformarán los datos de entrada y salida para el entrenamiento del modelo. Este conjunto de Tokens conforman el vocabulario de nuestro modelo.

Posteriormente, realizamos el entrenamiento del modelo. Nuestro modelo consiste en una Red Neuronal LSTM (Long Short-Term Memory), la cual ha demostrado tener un mejor desempeño que una Red Neuronal Feedforward en el modelado de lenguajes (LM) [3]. Nuestro objetivo es entrenar esta red para la predicción de secuencias, específicamente la próxima palabra o elemento que debería tener el archivo HTML. Una Red Neuronal LSTM nos da la ventaja de que tiene un mejor desempeño ante palabras que se encuentran fuera del vocabulario, lo cual es muy importante para nuestro caso puesto que nuestro vocabulario es muy pequeño, limitado a solo los tokens de nuestro archivo HTML original.

En la configuración del modelo, se ajustan los hiperparámetros de la Red Neuronal y se ajusta un tamaño de ventana. Destacamos que utilizamos la función de activación "ReLU", la función de activación de salida "softmax", un batch_size de 32. Sin embargo, aún tenemos que trabajar en el ajuste de los hiperparámetros.

El tamaño de ventana consiste en un número *n* de palabras antes o después de la palabra que buscamos predecir. Este parámetro en particular nos parece que puede tener una influencia muy significativa en el funcionamiento del modelo, pues nos podría determinar qué tanto nivel de cambio en el contexto del archivo HTML puede ser correctamente reconocido en el modelo. Por ejemplo, si cambian tanto los atributos como las etiquetas padre que preceden a la palabra que buscamos predecir, el tamaño de ventana debería ser lo suficientemente grande como para abarcar la cantidad de cambios. Sin embargo, esto también podría depender del tamaño del archivo, pues un tamaño de ventana muy grande para un archivo pequeño, podría afectar negativamente el rendimiento del modelo.

Para el proceso de entrenamiento tomamos *n* palabras (determinados por el tamaño de ventana) y como salida esperada tomamos a la palabra siguiente. Seguimos el mismo procedimiento hasta recorrer todas las palabras del archivo.

Con el modelo ya entrenado, nos queda nuestro último paso: construcción del archivo corregido.

Este paso consiste en tokenizar el archivo nuevo (el que no funciona) y utilizar los tokens como entrada del modelo.

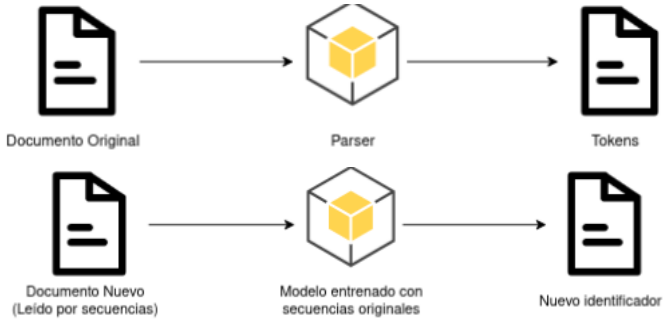

Fig. 1 Representación del producto

REFERENCIAS

[1] Filipe Guédes Venâncio and Ronaldo dos Santos Mello. 2020. A Similarity Function for HTML Lists. In Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia '20). Association for Computing Machinery, New York, NY, USA, 309–316. https://doi.org/10.1145/3428658.3430963

[2] K. Griazev and S. Ramanauskaitė, "HTML Block Similarity Estimation," 2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE), Vilnius, Lithuania, 2018, pp. 1-4, doi: 10.1109/AIEEE.2018.8592241

[3] Sundermeyer, Martin, Hermann Ney, and Ralf Schlüter. "From Feedforward to Recurrent LSTM Neural Networks for Language Modeling." IEEE/ACM Transactions on Audio, Speech, and Language Processing 23.3 (2015): 517-29. Web.