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

Daniel Artavia y 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 el localizador nuevo 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. PRODUCTO

Nuestro producto es un modelo que recibe como entrada la versión anterior (que pasa la prueba) del archivo HTML y la versión actual (que no pasa la prueba) del archivo HTML. La versión anterior se utiliza para entrenar al modelo de forma que pueda encontrar los cambios en los localizadores que ocasionan que la versión actual falle.

Una vez que se corrige a la versión actual, se desecha el modelo previamente entrenado y esta versión pasa a ser la versión antigua, por lo que se utilizará para entrenar a un nuevo modelo que servirá para encontrar los cambios en los localizadores de una futura versión actual del archivo. Es decir, por cada nueva versión del archivo se creará un modelo que servirá para resolver los conflictos de una futura versión del archivo. Además, cada archivo HTML tendrá un modelo que sirve exclusivamente para resolver los conflictos de sus futuras versiones (un modelo no sirve para varios archivos).

Nuestro producto consiste en 3 etapas: tokenización, entrenamiento del modelo y construcción del archivo corregido. La tokenización consiste en extraer información de los archivos HTML y transformarla en tokens. Los tokens que se obtienen del archivo anterior conforman el vocabulario con el que entrenaremos al modelo, mientras que, los tokens que se obtienen del archivo actual se utilizarán como entrada del modelo (ya entrenado) para encontrar los localizadores que cambiaron. Es importante saber que este proceso no elimina duplicados, es decir, nos limitamos a transformar el archivo a los tokens sin perder ninguna palabra, dando como resultado a una versión "tokenizada" del archivo, por decirlo de alguna manera. Además, los tokens son exclusivamente formados por la información que nos da valor: nombres de las etiquetas, etiquetas de cierre y los atributos de "class" e "id" (localizadores).

Antes de proceder con la descripción del entrenamiento, una aclaración: el modelo es una Red Neuronal LSTM, decisión que explicaremos en la sección de metodología Teniendo esto en consideración, el entrenamiento consiste en tomar el vocabulario mencionado anteriormente para entrenar al modelo. La forma de hacerlo consiste en recorrer los tokens en secuencias de "ventanas". Una ventana consiste en un número *n* de palabras que, en nuestro caso van antes y después de la palabra objetivo, por lo que es análogo a las ventanas del modelo CBOW de Word2Vec [4]. Entonces, en el entrenamiento, las ventanas alrededor de la palabra objetivo son la entrada de la red neuronal y la palabra objetivo es la salida esperada por la red. Las ventanas se corren una posición y cambian tanto las entradas como la salida de la red neuronal. Esto se hace hasta que se recorren todos los tokens, dando fin al entrenamiento del modelo.

Finalmente, con el modelo ya entrenado, se utiliza al archivo actual como entrada y obtenemos a los identificadores que fueron modificados y causan los fallos en las pruebas. Dando como resultado a la etapa de construcción del archivo corregido.

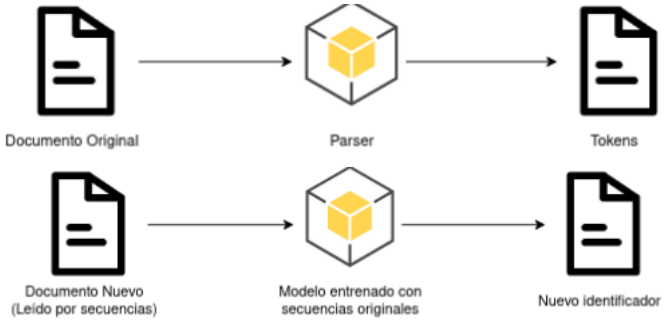

Fig. 1 Representación del producto

IV. METODOLOGÍA

En esta sección explicaremos el por qué de nuestras decisiones de diseño y nuestra metodología de experimentación.

A. Detalles de Implementación

Para la tokenización se utilizó BeautifulSoup, una biblioteca utilizada para Web Scraping de Python. Este proceso consiste en extraer todas las etiquetas y los atributos que nos interesan (clases e identificadores) de un archivo HTML.

Como adelantamos anteriormente, 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. Para los hiperparámetros, utilizamos los que recomienda Tensorflow por defecto para las redes LSTM.

Por otra parte, el tamaño de ventana es un parámetro que tiene 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. Por ello, decidimos ajustar este parámetro conforme íbamos realizando la experimentación.

B. Metodología de Experimentación

Para la experimentación, primero tomamos el HTML de una página web de la Universidad. Luego, aleatoriamente, tomamos un localizador y lo cambiamos. Este localizador es el que nuestro modelo debería detectar. Además, tomamos un "contexto" que podría definirse como las palabras/tokens alrededor del localizador que modificamos. Dentro de este contexto, también aleatoriamente, cambiamos una cantidad definida de palabras en posiciones aleatorias. Esto para ver si el modelo puede detectar cambios de localizadores cuando no solo cambió el localizador sino que también se modificó su contexto. La cantidad de cambios aleatorios en el contexto lo llamaremos "nivel de cambio".

Los parámetros que modificamos durante la experimentación fueron el tamaño de la ventana y el nivel de cambio. Se realizaron 3 réplicas de cada prueba, tomando como resultado al promedio de estas. Las corridas se realizaron de la siguiente manera:

- Para un tamaño de ventana 5 y un nivel de cambio entre 1-5, se realizaron 1159 pruebas.
- Para un tamaño de ventana 10 y un nivel de cambio igual a 1-10, se realizaron 1154 pruebas.
- Para un tamaño de ventana 20 y un nivel de cambio igual a 1-20, se realizaron 1144 pruebas.
- Para un tamaño de ventana 40 y un nivel de cambio igual a 1-40, se realizaron 1124 pruebas.

V. RESULTADOS

Para una ventana de tamaño 5, sin cambios en el contexto, el modelo tuvo un acierto del 92.4%. Con un solo cambio en el contexto, el modelo tuvo un acierto del 65.5% Ya con 2 cambios dentro del contexto, el modelo no llega a predecir ni la mitad de los casos de prueba. Todos los niveles de cambio se pueden ver en la figura 2.

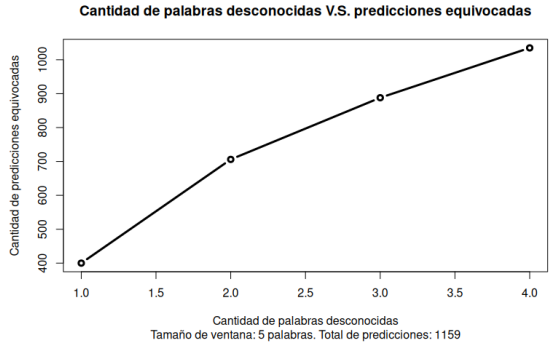

Fig. 2 Cantidad de predicciones con ventana de tamaño 5. El eje x representa el nivel de cambio. El eje y representa la cantidad de predicciones equivocadas.

Para una ventana de tamaño 10, con un solo cambio en el contexto, el modelo tuvo un acierto del 83.8%, porcentaje significativamente mayor que con una ventana de tamaño 5. Con 2 cambios dentro del contexto, el modelo tuvo un acierto del 68.9%, por lo que tuvo mayor precisión que el modelo con ventana de tamaño 5 cuando solo ocurría un cambio. Con 3 cambios el modelo no llega a predecir ni la mitad de los casos de prueba. Todos los niveles de cambio se pueden ver en la figura 3.

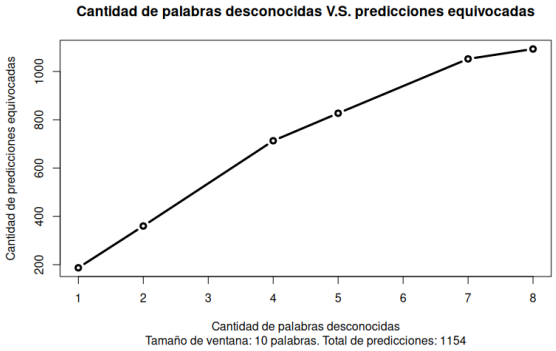

Fig. 3 Cantidad de predicciones con ventana de tamaño de 10. El eje x representa el nivel de cambio. El eje y representa la cantidad de predicciones equivocadas.

Para una ventana de tamaño 20, con un solo cambio en el contexto, el modelo tuvo un acierto del 91.4%. Con 2 cambios dentro del contexto, el modelo tuvo un acierto del 85.2%, y con 4 cambios el modelo tuvo un acierto del 71.8%, por lo que superó en precisión a los 2 modelos anteriores con ventanas de tamaño 5 y 10, cuando solo ocurrían uno o dos cambios. Ya con 8 cambios dentro del contexto, el modelo no llega a predecir ni la mitad de los casos de prueba. Todos los niveles de cambio se pueden ver en la figura 4.

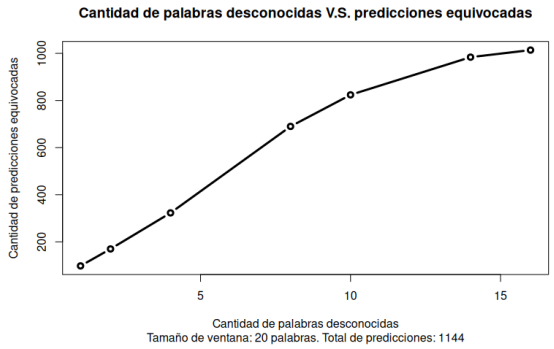

Fig. 4 Cantidad de predicciones con ventana de tamaño de 20. El eje x representa el nivel de cambio. El eje y representa la cantidad de predicciones equivocadas.

Para una ventana de tamaño 40, con un nivel de cambio de hasta 4 palabras, se tiene un peor acierto que utilizando una ventana de tamaño 20, con un 70.4%. Sin embargo, el porcentaje de acierto es superior con mayores niveles de cambio, por ejemplo, con 8 palabras desconocidas el porcentaje de acierto es del 58.3 y con 16 el porcentaje de acierto es de un 35.5%, menos de la mitad. Todos los niveles de cambio se pueden ver en la figura 5.

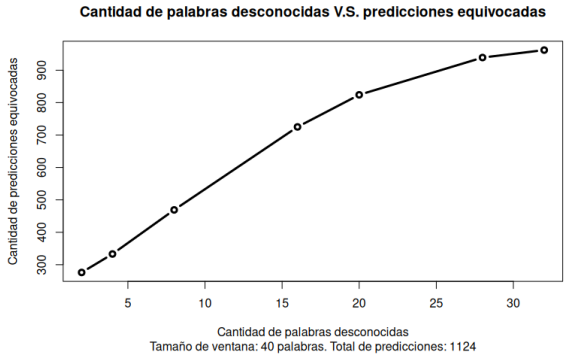

Fig. 5 Cantidad de predicciones con ventana de tamaño de 40. El eje x representa el nivel de cambio. El eje y representa la cantidad de predicciones equivocadas.

VI. CONCLUSIONES

Los resultados obtenidos de la experimentación nos demostraron que el nivel de cambio afecta la capacidad del modelo para predecir correctamente el elemento faltante. Además, se consigue observar que con un nivel de cambio menor o igual a 20% se obtiene la mayor probabilidad de predecir el elemento objetivo, mientras que porcentajes mayores aumentan rápidamente la probabilidad de errores.

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.
[4] Mikolov, Tomas & Chen, Kai & Corrado, G.s & Dean, Jeffrey. (2013). Efficient Estimation of Word Representations in Vector Space. Proceedings of Workshop at ICLR. 2013.