UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Modelos evolutivos en cáncer: mejoras y extensiones a web apps interactivas.

Autor: Laurentiu Mihai Adetu
Tutor: Ramón Díaz Uriarte
Ponente: Gonzalo Martínez Muñoz

junio 2023

Laurentiu Mihai Adetu
Modelos evolutivos en cáncer: mejoras y extensiones a web apps interactivas.

Laurentiu Mihai Adetu
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

En el desarrollo de cualquier proyecto de software informático, la fase de pruebas es fundamental para garantizar la calidad y el funcionamiento del producto final. Sin embargo, en ocasiones esta fase se descuida o se realiza de forma manual e ineficiente. Para facilitar la realización de pruebas efectivas y eficientes, existen diversas herramientas que permiten automatizar y simplificar esta tarea. Sin embargo, muchas de estas herramientas requieren un esfuerzo adicional por parte de los desarrolladores para adaptarse a las características específicas de cada proyecto.

En este trabajo de fin de grado se ha diseñado e implementado una herramienta para la automatización de pruebas sobre una aplicación web desarrollada utilizando *Shiny* [1], un paquete de *R* [2], que nos permite crear aplicaciones web directamente sobre una aplicación en este lenguaje, haciendo uso de la herramienta *Selenium WebDriver* [3], un estándar *W3C* [4] que permite controlar un navegador de forma remota. La aplicación web sobre la que vamos a utilizar la herramienta es **EvAM-Tools** [5], una aplicación científica para modelos evolutivos en cáncer.

Dado que *Selenium WebDriver* no está preparado para interactuar de manera intuitiva con todos los elementos de esta interfaz, ni manejar la actualización en tiempo real que presenta la aplicación, ha sido el objetivo de este proyecto crear una herramienta que facilite la automatización de pruebas para validar el funcionamiento de la aplicación que nos ocupa, consiguiendo una cobertura total de la interfaz web, tolerante a cambios en su estructura y diseño y una configuración personalizada para cada una de las pruebas que se vayan a ejecutar.

El objetivo final de la herramienta es aportar agilidad, seguridad y eficiencia al proceso de verificación de cambios realizados por los nuevos desarrollos sobre la aplicación, creando una herramienta capaz de simular los casos de uso y las acciones posibles sobre la interfaz de la aplicación web, así como validar el funcionamiento de sus elementos.

Palabras clave

Fase de pruebas, Shiny, R, Selenium WebDriver, W3C, simular, automatización de pruebas

Abstract

In the development of any computer software project, the testing phase is crucial to ensure the quality and functionality of the final product. However, sometimes this phase is neglected or carried out manually and inefficiently. To facilitate the execution of effective and efficient tests, there are various tools available that allow for automation and simplification of this task. However, many of these tools require additional effort from developers to adapt to the specific characteristics of each project.

In this final degree project, a tool has been designed and implemented for test automation on a web application developed using *Shiny* [1], a *R* [2] package, which enables the creation of web applications directly within this language. The tool makes use of *Selenium WebDriver* [3], a *W3C* [4] standard that allows remote control of a browser. The web application on which we will use the tool is **EvAM-Tools** [5], a scientific application for evolutionary models in cancer.

Since *Selenium WebDriver* is not designed to intuitively interact with all elements of this interface or handle real-time updates presented by the application, the objective of this project has been to create such functionality for seamless integration with the application at hand. This includes achieving complete coverage of the web interface, being tolerant to changes in its structure and design, and providing customized configuration for each test to be executed.

The ultimate goal of the tool is to bring agility, security, and efficiency to the process of verifying changes made to the application prior to its public use. It aims to create a tool capable of simulating use cases and possible actions on the web application's interface, as well as validating the functionality of its elements.

Keywords

Testing phase, Shiny, R, Selenium WebDriver, W3C, simulating, test automation

Índice

Lista de figuras

1

Introducción

1.1. Motivación

La calidad y el correcto funcionamiento de una aplicación web son requisitos indispensables para garantizar la satisfacción de los usuarios y el éxito del proyecto. Sin embargo, cuando se trata de una aplicación científica que involucra a varios desarrolladores, el desafío es aún mayor. **EvAM-Tools** es una aplicación web científica que pone a disposición de sus usuarios diversas herramientas para rápidamente construir, manipular y explotar los modelos MPC (Modelos de progresión del cáncer) a través de una interfaz gráfica de usuario. La unión entre la informática y otras ciencias ha supuesto un avance en los estudios de las mismas como nunca antes se ha visto, no obstante es en estos proyectos que juntan ramas como la medicina, donde se debe insistir en asegurar un correcto funcionamiento junto a unos resultados fiables.

Según lo expuesto en el párrafo anterior, el desarrollo de un sistema de validación que; pruebe el correcto funcionamiento de los elementos presentes en la aplicación después de cada nuevo desarrollo, sea capaz de identificar errores y permitir la automatización de pruebas, resulta ser una parte esencial de las actividades de aseguramiento de calidad de un software. Aportando claridad sobre el estado del producto desarrollado y asegurando un correcto funcionamiento.

Una parte fundamental para asegurar la calidad son las pruebas software. Estas pueden ser de varios tipos; unitarias, funcionales, de rendimiento, de integración, etc. Algunas, por su naturaleza, pueden ser automatizadas, lo que facilita su ejecución y nos asegura tener un proceso de validación completo.

Cada vez las aplicaciones, sobre todo las aplicaciones web, se vuelven más complejas; variando requisitos y sobre todo añadiendo, modificando o eliminando funcionalidades. Esto hace que la implementación de pruebas cobre una importancia mayor pero, realizar la validación de estas pruebas manualmente como es el caso puede resultar ineficiente. Debido a la gran variedad de aplicaciones, existe también una variedad de herramientas para su prueba; no obstante estas herramientas por regla general no se adaptan por completo a todas las necesidades de las aplicaciones. Una de las más completas y con más importancia es *Selenium* [6], un framework open-source para la automatización

de pruebas controlando navegadores web, nos proporciona acceso a una API versátil para reproducir casos de prueba tal y como lo haría el usuario final. Su integración con una gran variedad de lenguajes de alto nivel y la capacidad de agregar funcionalidad propia, adaptada a cada aplicación lo hace una opción a tener en cuenta para cualquier proyecto de automatización para aplicaciones web.

Este trabajo de fin de grado se ha desarrollado en torno a la aplicación EvAM-Tools, una aplicación que sigue creciendo y recibiendo nuevos desarrollos pero que, debe mantener una funcionalidad entre desarrollos. La importancia de mantener estos casos de uso generó interés por el desarrollo de una herramienta capaz de automatizarlos y facilitar futuros desarrollos, asegurando la funcionalidad de la aplicación.

1.2. Objetivos

El objetivo final del proyecto es la creación de una herramienta sencilla de entender y de utilizar para la prueba, automatización, aseguración de funcionalidad, tolerante a cambios, personalizada y configurada para funcionar sobre la interfaz web de **EvAM-Tools**.

Para ello, se llevará a cabo un estudio sobre los conceptos básicos relacionados con la calidad del software y su importancia en el proceso de desarrollo disponibles para desarrollar una herramienta capaz de probar; la funcionalidad de todos los elementos actualmente presentes en la aplicación web y la validación de todos los casos de uso disponibles. También servirá de guía para futuras implementaciones de nuevos elementos de una forma intuitiva y con la mínima interacción con el código más allá de la propia creación de las pruebas.

En general, se busca:

- Realizar un estudio sobre los conceptos básicos relacionados con la calidad del software y su importancia en el proceso de desarrollo.
- Introducir la aplicación sobre la que vamos a diseñar la herramienta.
- Valorar posibles tecnologías útiles para el desarrollo de la herramienta.
- Diseñar una herramienta capaz de interactuar con todos los elementos y capaz de completar todos los casos de uso necesarios sobre EvAM-Tools cubriendo las carencias de las herramientas existentes.
- Crear un manual de referencia para futuros desarrollos.
- Validar el potencial de la propia herramienta de automatización, analizando beneficios e inconvenientes de la automatización de dichas pruebas.

2

Estado del arte

2.1. Descripción de las pruebas de interfaces web y su importancia en el desarrollo de aplicaciones web

Las pruebas de interfaces web son fundamentales en el desarrollo de aplicaciones web, permitiéndonos validar la funcionalidad y asegurar un buen uso del producto desarrollado. Estas pruebas contribuyen a ofrecer una interfaz de calidad, confiable y satisfactoria para los usuarios finales.

La importancia de las pruebas de interfaces web radica en varios aspectos:

- **Garantizar la funcionalidad**: estas pruebas permiten verificar que todas las características y funcionalidades de la aplicación web, como formularios, botones, elementos interactivos, etc, funcionen correctamente y realicen la acciones a las que están destinados. Estas pruebas pueden exponer errores en la acción combinada de estos elementos y validar la calidad del producto final.
- **Mejorar la usabilidad**: se permite evaluar la facilidad de uso y la experiencia del usuario mediante estas pruebas. Se verifican aspectos como la navegación intuitiva, la claridad de la información presentada y la retroalimentación adecuada
- **Asegurar la compatibilidad**: las aplicaciones deben ser compatibles con diferentes navegadores (por ejemplo Chrome, Firefox, Safari) debido a la gran diversidad de plataformas utilizadas por los usuarios.
- **Detectar y corregir errores**: Las pruebas de interfaces web ayudan a identificar y solucionar errores o problemas técnicos que pueden surgir durante el desarrollo de la aplicación como enlaces rotos, formularios que no envían datos correctamente, problemas de rendimiento ente otros.
- **Ahorrar tiempo y costes**: realizando estas pruebas de manera regular y sistemática ayuda a identificar problemas en fases tempranas del desarrollo, lo cual reduce el coste y tiempo requerido para su corrección más adelante.
- **Mantenibilidad y evolución**: la realización de pruebas de manera continuada en el tiempo a medida que se introducen nuevos cambios, nos asegura que la aplicación siga funcionando correctamente a lo largo del tiempo.

2.1.1. Introducción al entorno tecnológico de EvAM-Tools

EvAM-Tools es una aplicación científica desarrollada en *R*, con una interfaz para el usuario final creada con *Shiny*, para la creación y explotación de modelos de progresión del cáncer como: Árboles Oncogenéticos (Oncogenetic Trees - OT), Redes bayesianas conjuntivas (Conjuntive Bayesian Networks - CBN), Redes causales ocultas extendidas de Suppes-Bayes (Hidden Extended Suppes-Bayes Causal Networks - H-ESBCN), OncoBN, Redes de peligro mutuo (Mutual Hazard networks - MHN), Gráfico acíclico dirigido (Directed Acyclic Graph - DAG).

Estos modelos [7] son interactuados mediante una interfaz creada usando *Shiny* donde, cualquier cambio en los paquetes de la aplicación en *R* puede tener consecuencias en la interfaz afectando a su funcionalidad. Por eso, es muy importante tener una forma de probar automáticamente que estos cambios no han afectado negativamente a la aplicación web. La validación de estos modelos se realiza manualmente siguiendo los casos de uso presentes en el documento *EvAM-Tools: Examples* [8], donde se detallan toda la funcionalidad a disposición del usuario final y la manera de construir dichos modelos siguiendo diversas estrategias como:

- **Inferencia de los modelos a partir de datos subidos desde un fichero**: esta manera de construcción de modelos debe permitir al usuario seleccionar un fichero de su equipo con los datos de generación preparados y construir un modelo a partir de ellos.
- **Introducción de datos sintéticos construidos manualmente**: la selección del número de genes, los genotipos y su cantidad y el nombre de los datos se introducen de manera manual por parte del usuario.
- **Construcción de modelos MPC (DAG con sus tasas/probabilidades y modelos MHN) y simulación de datos sintéticos a partir de ellos**: en estos apartados el usuario debe ser capaz de modificar el modelo pertinente con los datos que necesite de manera manual, editando los elementos preestablecidos de cada modelo.

Cada una de estas estrategias presenta funcionalidad compartida entre ellas, al igual que únicas para cada modelo a disposición del usuario en una interfaz clara mostrada sobre una aplicación web con carga de contenido dinámico. Para poder probar todos los elementos involucrados en estas estrategias, se debe cerciorar de la correcta carga de todos los elementos involucrados en ellas, así como su funcionamiento individual y el colectivo lo cual, ha resultado ser un desafío para la implementación de las pruebas debido al desfase entre la carga de los elementos por parte de la aplicación y la velocidad de ejecución de las pruebas sobre la interfaz.

Siendo esta una aplicación web científica la comprobación de los resultados y lo elementos debe ser más estricta. La realización de pruebas de manera manual por todos los posibles casos de uso puede resultar tediosa, por lo que disponer de una herramienta especialmente diseñada para la aplicación que estamos tratando es una ayuda al desarrollo y mantenimiento muy importante.

2.2. Aplicaciones y herramientas similares

Existen varias tecnologías implementadas en aplicaciones para la prueba de interfaces. Estos son unos ejemplos de las más populares y más utilizadas por aplicaciones ya existentes.

- Selenium [6]: Selenium es una herramienta de prueba de interfaz web ampliamente utilizada y de código abierto. Permite la automatización de pruebas en navegadores web reales, lo que significa que se pueden simular acciones del usuario y verificar el comportamiento de la aplicación en diferentes escenarios. Selenium ofrece soporte para múltiples lenguajes de programación, lo que permite escribir scripts de prueba flexibles y personalizados. Se describe en más detalle en las secciones que siguen.
- Puppeteer [9]: Puppeteer es una biblioteca de Node.js desarrollada por Google que proporciona un control de alto nivel para el navegador Chrome. Permite la automatización de tareas en el navegador, como interactuar con elementos de la interfaz, generar capturas de pantalla y realizar pruebas de rendimiento. Puppeteer también es compatible con Python y ofrece una API fácil de usar para escribir scripts de prueba.
- Playwright [10]: Playwright es otra biblioteca de automatización de navegadores desarrollada por Microsoft. Al igual que Puppeteer, ofrece un control de alto nivel sobre los navegadores web, incluidos Chrome, Firefox y Safari. Playwright es multiplataforma y es compatible con varios lenguajes de programación, incluido Python. Proporciona capacidades avanzadas, como pruebas en paralelo, emulación de dispositivos y grabación de tráfico de red.
- Cypress [11]: Cypress es una herramienta de prueba de interfaz web que se centra en la simplicidad y la velocidad. Permite escribir pruebas en JavaScript y ejecutarlas directamente en el navegador, lo que facilita la depuración y la interacción con la aplicación durante las pruebas. Cypress ofrece una amplia gama de funcionalidades, como la grabación y repetición de pruebas, la inspección de elementos y la capacidad de realizar aserciones y validaciones en tiempo real.
- Protractor [12]: Protractor es una herramienta de prueba de interfaz de usuario específicamente diseñada para aplicaciones AngularJS y Angular. Está basada en WebDriver y permite realizar pruebas automatizadas en aplicaciones web construidas con estos frameworks. Protractor ofrece funcionalidades específicas para lidiar con el modelo de ejecución asíncrono de AngularJS y Angular, lo que facilita la escritura y ejecución de pruebas.
- Appium [13]: Appium es una herramienta de automatización de pruebas para aplicaciones móviles. Permite realizar pruebas en dispositivos reales y emuladores tanto en plataformas iOS como Android. Appium proporciona una API consistente para interactuar con las aplicaciones, lo que permite escribir scripts de prueba en lenguajes populares como Python, Java, C#, entre otros.
- Winium [14]: Winium es una herramienta de automatización de pruebas para aplicaciones de Windows basada en Selenium WebDriver. Está diseñada específicamente para automatizar prue-

bas en aplicaciones de escritorio de Windows. Winium ofrece soporte para diferentes tecnologías de Windows, como WinForms, WPF y aplicaciones de la Tienda Windows (Windows Store).

- WinAppDriver [15]: WinAppDriver es una herramienta desarrollada por Microsoft para automatizar pruebas en aplicaciones de Windows. Permite interactuar con elementos de la interfaz de usuario de aplicaciones de Windows mediante lenguajes de programación y bibliotecas populares como Selenium y Appium. WinAppDriver se utiliza principalmente para realizar pruebas de regresión en aplicaciones de Windows.
- MS UI Automation [16]: MS UI Automation es una API proporcionada por Microsoft para automatizar pruebas en aplicaciones de Windows.Permite interactuar con elementos de la interfaz de usuario, obtener información sobre ellos y realizar acciones como clics, ingreso de texto, etc. MS UI Automation se utiliza principalmente en aplicaciones de escritorio de Windows y se puede integrar con herramientas de automatización de pruebas.
- TestComplete [17]: Es una herramienta de prueba funcional y automatización de pruebas desarrollada por SmartBear. Ofrece capacidades para realizar pruebas en aplicaciones de escritorio, web y móviles. TestComplete permite grabar, editar y reproducir pruebas, así como también ofrece funciones de scripting y una variedad de opciones de informes.

2.3. Tecnologías escogidas para el desarrollo

Tras el análisis de las herramientas disponibles en el apartado 2.2 para la ejecución de pruebas sobre interfaces web, se ha observado cómo ninguna de las tecnologías descritas satisface todos los requisitos necesarios para nuestro caso concreto el cual pretende la prueba de elementos de forma individual así como en conjunto, presentes en la aplicación EvAM-Tools.

Además, debido a que la herramienta se utilizará sobre una web generada utilizando *Shiny* [1] la cual actualiza en tiempo real los elementos de la interfaz de acuerdo con las acciones del usuario y reorganiza la estructura de la propia interfaz a medida que se avanza en el flujo de trabajo, necesitamos un desarrollo más allá del simple uso de las funciones nativas de las tecnologías existentes. Debido a ello, se necesita contar con una herramienta base capaz de comunicarse con un lenguaje de programación de alto nivel con suficiente versatilidad y funcionalidad disponible. Por ello se ha elegido *Selenium WebDriver* en su versión 3.141.0 como aplicación base sobre la que trabajar. Para asegurarnos el correcto funcionamiento de la aplicación web desplegada, debemos ser capaces de probar todos y cada uno de los elementos por separado así como la combinación de los mismos y analizar el comportamiento ante ciertos flujos de trabajo, algunos más, otros menos parecidos un uso real que se le va a dar a la aplicación.

Selenium WebDriver es ampliamente considerado como uno de los mejores frameworks para pruebas automatizadas sobre páginas web con elementos complejos por varias razones entre las cuales

destacan: la amplia gama de lenguajes de programación de alto nivel con los que podemos ejecutar el framework incluyendo *Java*, *Python*, *C#*, *Ruby* y *JavaScript* entre otros; la flexibilidad y expansibilidad, amplia comunidad y recursos disponibles; y una interacción con todos los elementos de una web debido a que ofrece una variedad de métodos y funciones para interactuar con elementos de la interfaz, incluidos elementos emergentes, elementos dinámicos y elementos dentro de iframes lo que permite simular una gran cantidad de acciones del usuario final.

Como se ha indicado antes, se necesitaba una herramienta capaz de comunicarse con lenguajes de alto nivel. Entre las opciones disponibles que ofrece *Selenium WebDriver*, *Python* (3.8.10) [18], se ha considerado como la mejor opción para agregar un procesamiento adicional sobre la funcionalidad que *Selenium WebDriver* puede ejecutar sobre una interfaz web como por ejemplo: recopilación y generación de informes de errores. Debido a su popularidad, sencillez y fácil portabilidad e implementación, es uno de los lenguajes más populares entre desarrolladores y profesionales de otros sectores que necesitan de la programación en su dia a dia, siendo este el inicio de una herramienta que evolucionará con el avance de la aplicación web y necesaria la integración de nuevas pruebas, las alternativas con las que *Selenium WebDriver* puede trabajar pierden el atractivo incluso con las ventajas de rendimiento y robustez.

El desarrollo de las pruebas se ha hecho sobre el navegador *Firefox v102.4esr* [19], donde las siglas ESR representan Extended Support Release (versión con ciclo de asistencia extendido) asegurándonos así una larga vida útil antes de la necesidad de cambiar el navegador y posiblemente adaptar funcionalidad. Junto con este navegador se ha hecho uso de un motor de navegación, *Geckodriver v0.23.0* [20], diseñado por Mozilla Corp. [21] para hacer de enlace entre el navegador y *Selenium WebDriver*. Siendo *Chrome* el navegador más popular en la actualidad como podemos observar en la figura 2.1 sacada directamente de *W3Counter* [22], se ha escogido Firefox debido a su menor uso de recursos, siendo uno de los objetivos mantener la herramienta eficiente. No obstante la funcionalidad es independiente del buscador sobre el que se ejecuten las pruebas y la herramienta puede ser configurada para funcionar con los navegadores más populares que permitan la integración con *Selenium WebDriver*.

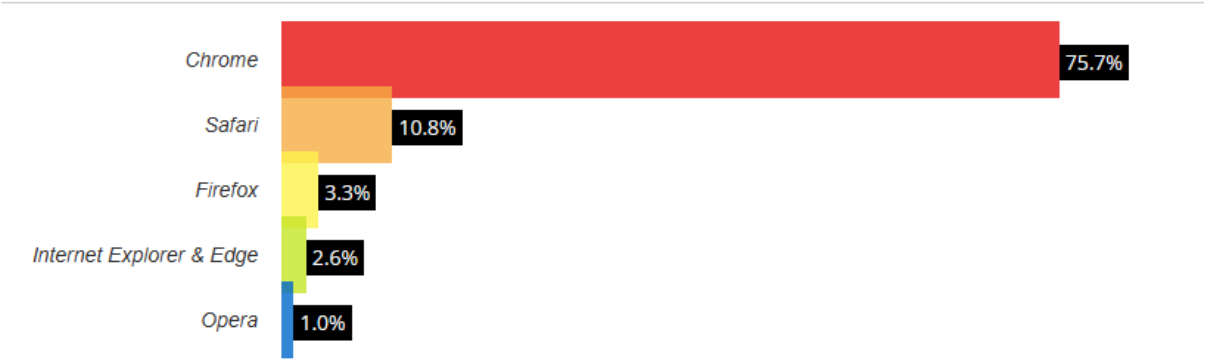

Figura 2.1: Gráfico de buscadores más utilizados en 2023

3

Diseño

A grandes rasgos, la herramienta a desarrollar deberá presentar un diseño modular en todos los apartados posibles para facilitar el crecimiento de la misma sin la necesidad de una refactorización entera de código.

Así mismo el mecanismo de enlace, necesario para juntar los casos de prueba desarrollados, debe poder controlar el comportamiento completo de la herramienta de acuerdo a las necesidades de la ejecución. Este mecanismo es capaz de controlar la creación de una ventana visible para el desarrollador, seleccionar el tamaño de la ventana, guardar la traza de la ejecución en un informe para su posterior análisis, guardar capturas de la ejecución en caso de error, entre otros que se describirán a lo largo del desarrollo del diseño.

Con esta información y los detalles introducidos en capítulos previos, se pueden detallar y describir los requisitos que la herramienta ha de cumplir.

3.1. Requisitos funcionales

RF-1.– Debe permitir el registro de las pruebas.

RF-2.– Debe ser capaz de ejecutar pruebas automatizadas en la interfaz web para verificar su funcionalidad.

RF-3.– Debe ser capaz de capturar y registrar los resultados de las pruebas, incluyendo errores.

RF-4.– Debe ser compatible con varios navegadores y versiones.

RF-5.– La herramienta debe ser compatible con navegadores de 64-bits.

RF-6.– Los navegadores soportados deben tener una versión de ciclo de asistencia extendido.

RF-7.– Debe ser capaz de manejar todos los elementos presentes en la interfaz web actualmente.

RF-8.– Debe poder ejecutarse con y sin interfaz gráfica, a petición del usuario.

RF-9.– Las pruebas creadas deben poder ejecutarse en grupos.

RF-10.– Las ejecuciones de grupos deben poder contener la misma prueba en varios grupos.

RF-11.– Debe ser compatible con un sistema Linux basado en las distribuciones más populares.

RF-12.– Debe ser capaz de ejecutar todas las pruebas incluso si una falla.

RF-13.– Debe avisar en caso de cualquier error descubierto previo a la ejecución de los tests que pueda comprometer la ejecución.

RF-14.– Debe poder utilizarse sobre la web desplegada en producción.

RF-15.– Debe poder utilizarse sobre un entorno de pruebas desplegado en local.

RF-16.– Cada función debe estar correctamente comentada y detallar su uso y justificar su necesidad.

RF-17.– Deben implementarse pruebas para el análisis de datos transversales con análisis de datos BRCA.

RF-18.– Deben implementarse pruebas para el análisis de datos transversales con cambio de tamaño para las variables.

RF-19.– Deben implementarse pruebas para el análisis de datos transversales con análisis de datos Ováricos.

RF-20.– Deben implementarse pruebas para el análisis de datos introducidos manualmente.

RF-21.– Deben implementarse pruebas de generación de datos a partir de modelos conocidos.

RF-22.– Deben implementarse pruebas de todos los elementos interactuables de la interfaz.

RF-23.– Deben implementarse pruebas para cubrir todos los casos de uso presentes en el documento `https://rdiaz02.github.io/EvAM-Tools/pdfs/evamtools_examples.pdf`

3.2. Requisitos no funcionales

RNF-1.– El uso debe ser sencillo, con instrucciones claras para los usuarios sobre su configuración y ejecución.

RNF-2.– La herramienta debe garantizar la seguridad de sus datos.

RNF-3.– Debe ser capaz de manejar un alto volumen de pruebas y adaptarse a entornos de prueba en constante crecimiento.

RNF-4.– Debe ser tolerante a cambios en la interfaz actual.

RNF-5.– Debe ser modular y fácil de mantener, permitiendo agregar nueva funcionalidad y realizar actualizaciones sin afectar a las pruebas existentes.

RNF-6.– La creación de grupos de pruebas ha de ser ser intuitiva y clara.

RNF-7.– Debe ser ligera y no consumir una gran cantidad de recursos del sistema.

3.3. Arquitectura de la aplicación

Para poder mantener un control sobre el mayor número de aspectos posibles sobre la ejecución, con la mayor claridad posible y sin la necesidad de crear un menú o una selección de parámetros individual a cada test, se ha desarrollado el siguiente esquema para la aplicación mostrado en la figura 3.1.

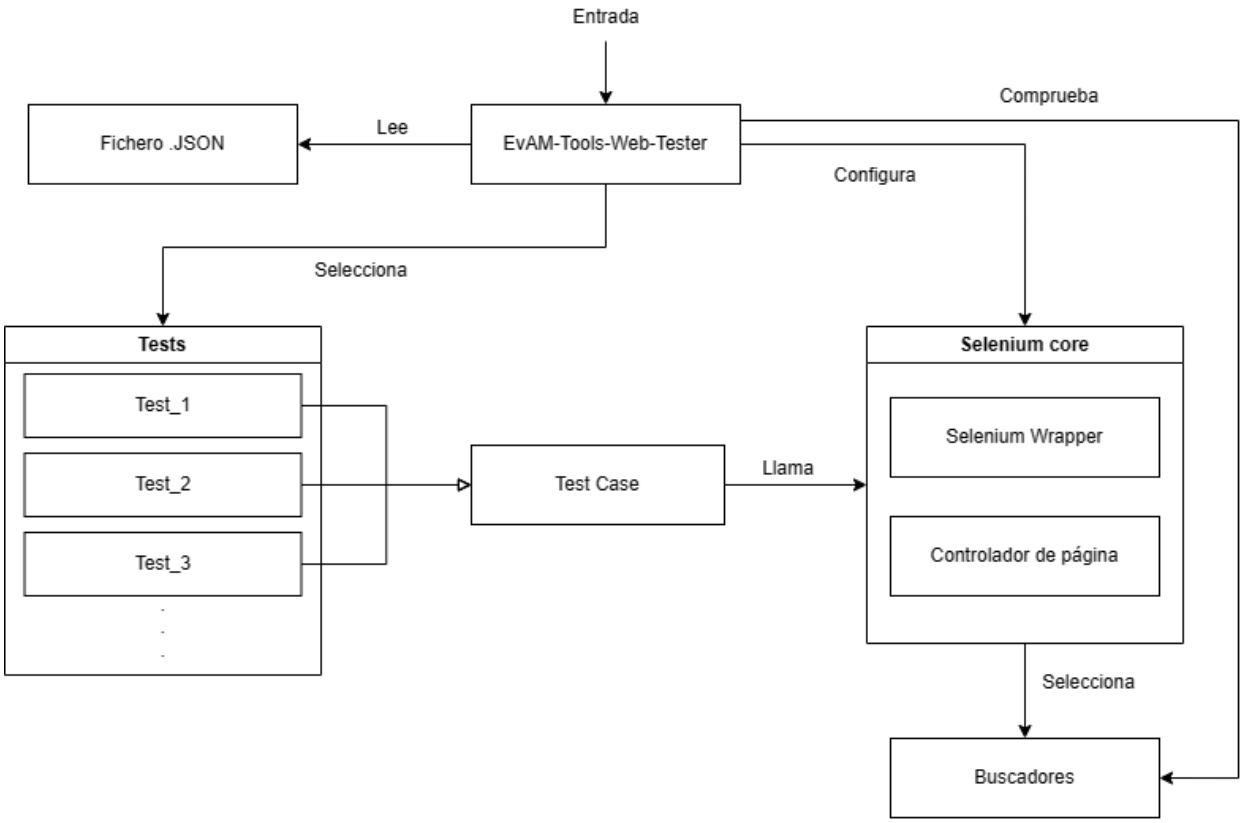

Figura 3.1: Arquitectura propuesta para la herramienta

Como se ha descrito anteriormente contamos con un mecanismo de enlace el cual hace de punto de entrada y coordinador de acciones entre los distintos elementos que contiene la herramienta, es el encargado de recibir los argumentos, validar la configuración planteada, comprobar las pruebas disponibles, ejecutarlos y guardar sus resultados para futuras revisiones.

3.3.1. Fichero .JSON

Comenzamos describiendo primero este fichero ya que, de cara a una ejecución real, este debe ser el primer fichero al que hacer referencia para determinar la ejecución que se quiere hacer.

Este fichero contendrá, en formato estandarizado json, varios grupos de pruebas identificadas por un nombre descriptivo del grupo. El número de grupos disponibles en el fichero no presenta un límite,

pudiendo tener tantos como diferentes grupos de prueba para diferentes situaciones o elementos a probar se quiera.

Cada grupo contendrá a su vez una estructura para indicar el nombre del test y los argumentos de cada uno, pudiendo personalizar cada test de manera independiente sin afectar a la ejecución de otros. Esto nos permite versatilidad para mejorar rendimiento o para focalizar tests más críticos y otros menos importantes en los que el comportamiento es más laxo frente a errores.

Frente a este diseño se soluciona también la necesidad de poder ejecutar una prueba que pueda probar una funcionalidad más general y pueda ser necesario en varios grupos sin la necesidad de duplicar su código en varios ficheros de pruebas: solo sería necesario hacerle referencia en tantos grupos como sea necesario.

3.3.2. EvAM-Tools-Web-Tester (Punto de entrada)

Este es el coordinador principal. este módulo, probablemente el más simple pero importante de todos, es el encargado de validar la configuración seleccionada y asegurar un correcto funcionamiento de la ejecución.

Las acciones realizadas por este módulo son por orden:

- Lectura del fichero .JSON apartado 3.3.1 para obtener las pruebas a ejecutar.
- Localizar dichas pruebas en el módulo 'tests' apartado 3.3.3; Notificando en caso de no estar disponible alguna de las pruebas sin parar la ejecución.
- Comprobación de los argumentos de configuración para cada una de las pruebas.
- Comprobación del estado de la instalación del buscador seleccionado para la ejecución. En caso de no estar disponible se finalizaría la ejecución, mostrando la configuración a realizar.
- Configuración del módulo Selenium Core apartado 3.3.5 para cada una de las pruebas.
- Ejecución de las pruebas.

3.3.3. Tests

Este módulo es un módulo dinámico de *Python* que facilita la creación y ejecución de pruebas de forma sencilla y flexible. El módulo se puede importar como un paquete y para agregar pruebas al mismo es suficiente con arrastrarlos a la carpeta del módulo. Las pruebas que se creen en este módulo pueden tener el nombre que el desarrollador prefiera, se ha optado por seguir la convención de empezar por *'test_'* para mantener cierto orden y similitud con otras herramientas de ejecución de tests. El hecho de que una prueba esté en este módulo no implica que se vaya a ejecutar automáticamente, ya que eso depende del fichero *config.json* que se encuentra en el directorio raíz del proyecto. De esta

forma, se puede tener una colección de pruebas de diferente complejidad y alcance en este módulo, sin que interfieran entre sí ni con el resto del código. Las funciones dentro de los tests al igual que el nombre deben seguir ciertas pautas discutidas en la implementación de los mismos.

3.3.4. Test Case

Las pruebas a desarrollar en el módulo anterior son todas clases las cuales, heredan de esta clase generalizada que contiene las funciones comunes de todas las pruebas, las cuales se encargan de la llamada del módulo *'Selenium Core'* apartado 3.3.5, aplicar la configuración obtenida de la entrada (EvAM-Tools-Web-Tester apartado 3.3.2), ejecutar la prueba y a continuación cerrar y recoger los datos de las pruebas. Estas acciones han sido las decididas como mínimas y necesarias, pudiéndose ampliar la funcionalidad individual de cada prueba de acurdo con sus necesidades.

3.3.5. Selenium Core

Hasta ahora la estructura descrita ha sido en torno a la aseguración del funcionamiento de la ejecución de las pruebas. Este módulo es el encargado del control y ejecución de las pruebas sobre el buscador seleccionado. Como se puede observar en la figura 3.1 este módulo consta de dos partes: Selenium Wrapper y el Controlador de página.

El módulo Selenium Wrapper es un módulo independiente que encapsula la llamada de los buscadores y el control sobre el mismo. Esta clase permite utilizar las funciones de Selenium de una manera sencilla y acotada, sin necesidad de intervenir mucho en el código. Esta es una de las partes más técnicas del desarrollo, ya que se encarga del levantamiento, localización y configuración de los buscadores. Por otro lado el módulo de control de página es la personalización y adaptación del framework de Selenium a las estructuras generadas por *Shiny* y en concreto a la interfaz que nos ocupa.

A la hora de diseñar el controlador se ha dividido por bloques, cada uno de estos bloques conteniendo parte de la funcionalidad de la propia interfaz de la aplicación. Las figuras figura 3.2, figura 3.3, figura 3.4 y figura 3.5 presentan la funcionalidad prestada por EvAM-Tools. Se describe, a grandes rasgos, la estructura seguida por *Shiny* a la hora de crear la interfaz; basándonos en esa estructura se ha planteado el desarrollo de la herramienta. A continuación se desglosa la funcionalidad en cada vista haciendo hincapié en los objetivos que se necesitan en cada una de ellas.

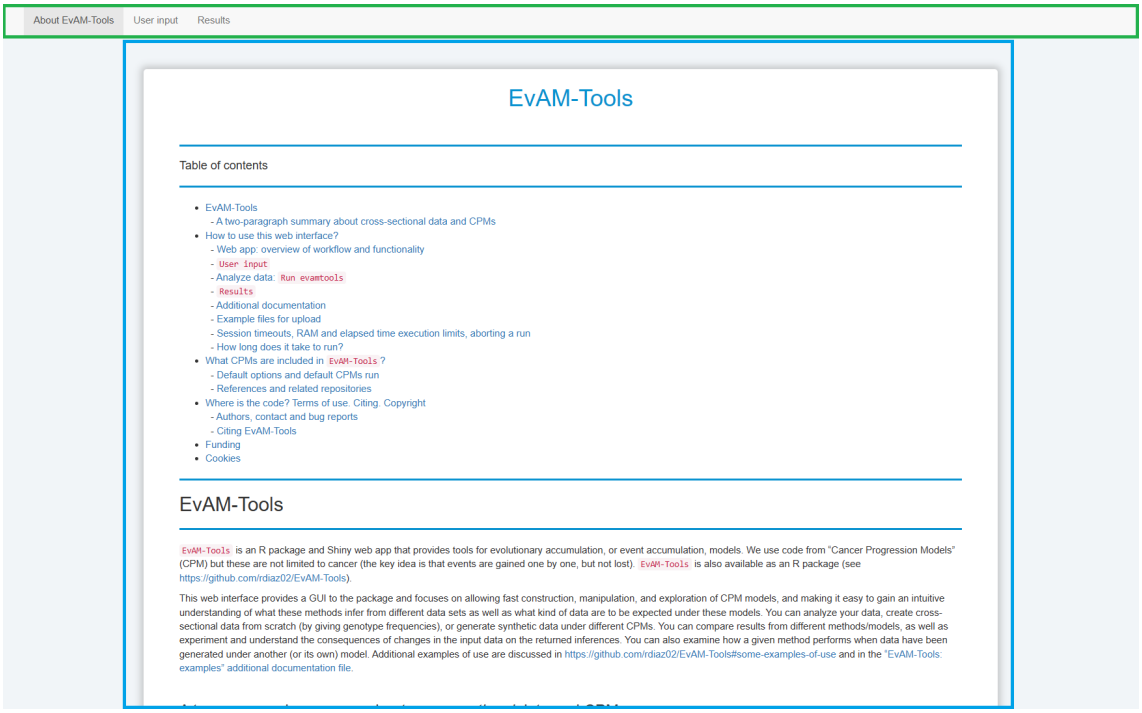

Figura 3.2: Vista interfaz About EvAM-Tools

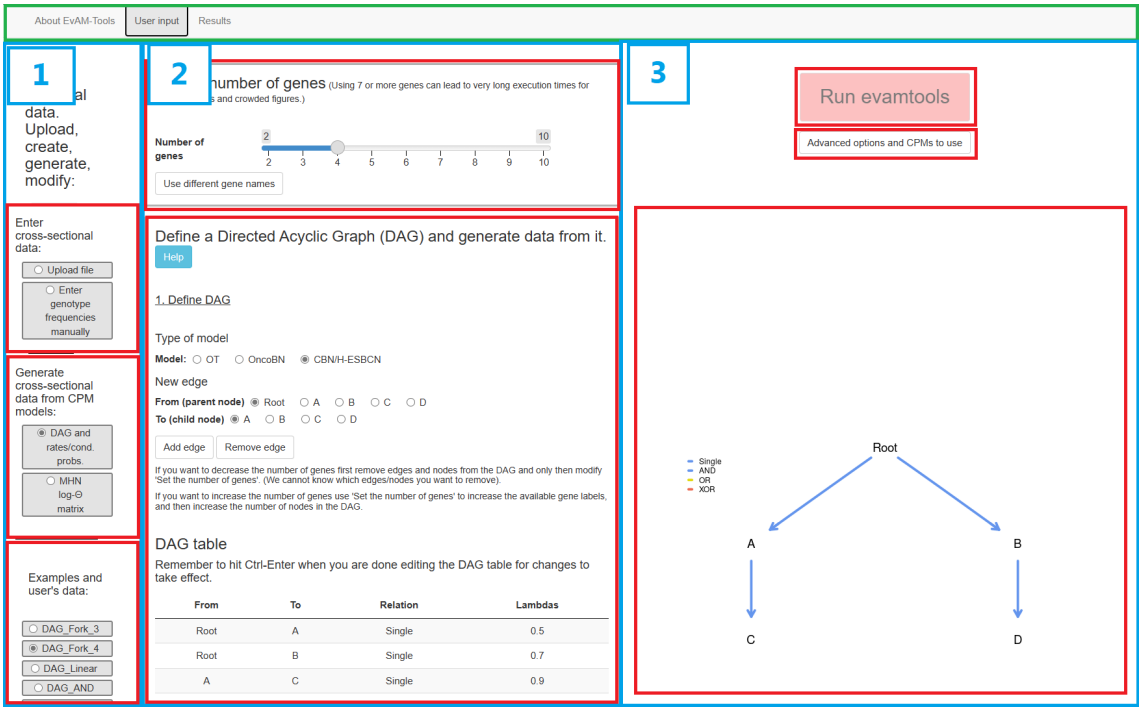

Figura 3.3: Vista interfaz User input

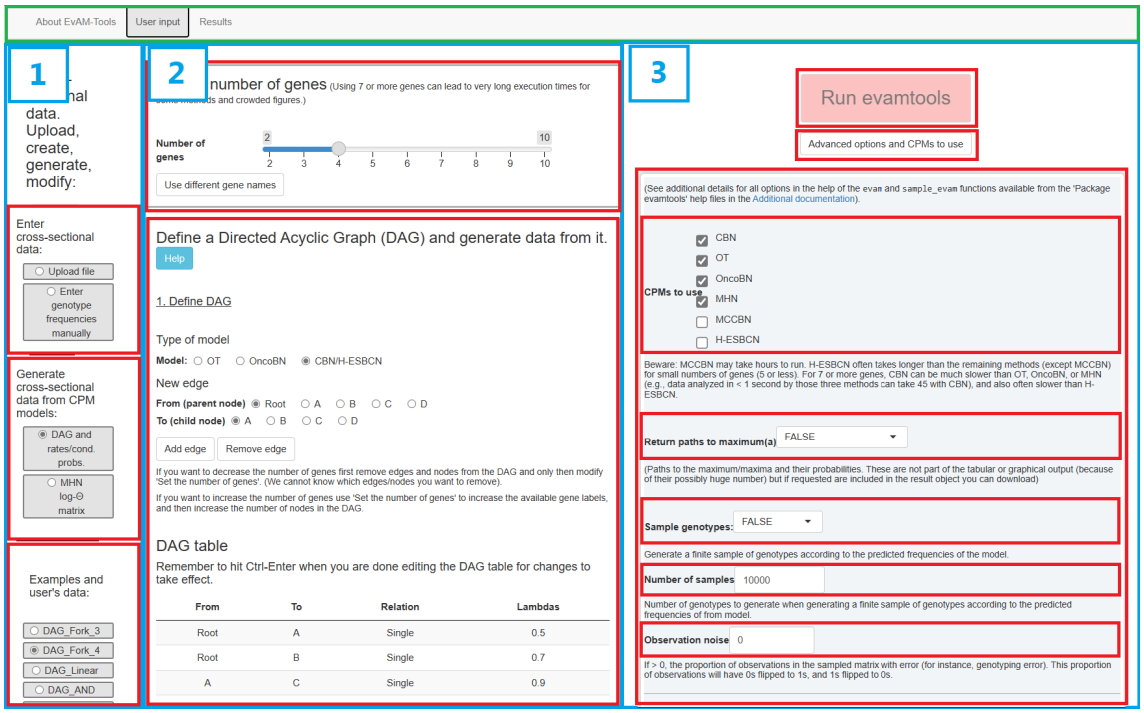

Figura 3.4: Vista interfaz User input con opciones avanzadas desplegadas

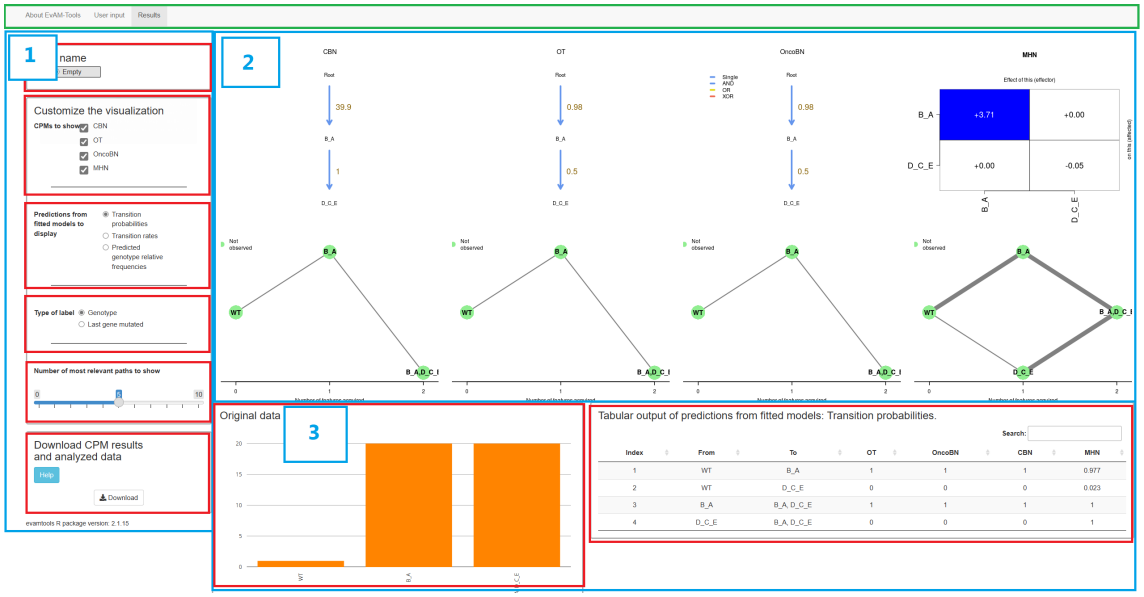

Figura 3.5: Vista interfaz Results

- En la figura 3.2: esta interfaz es la más simple. Los elementos interactuables han sido resaltados, el marco verde muestra la barra de navegación presente en todas las vistas y el marco azul muestra la información de esta vista de la interfaz la cual se concentra entera en un mismo elemento. Esta información consta de texto plano junto a enlaces de interés relacionados con el tema principal de la aplicación los cuales se prueba que todos sean funcionales y hagan referencia a su respectiva posición.
- En las figura 3.3 y figura 3.4: en estas vista se observan algunos elementos interactuables de la interfaz *'User input'*. Este ejemplo es la vista inicial que un usuario tiene sobre la web. Como hemos indicado antes, se pueden ver los bloques en los que se divide la vista de esta interfaz numerados, donde cada uno de estos bloques divide una parte de la funcionalidad, siendo el **bloque 1** el encargado de seleccionar el modelo y los datos con los que se trabaja, el **bloque 2** el encargado de recoger los datos introducidos por el usuario de la web para el modelo y el **bloque 3** el encargado de visualizar los datos, configurar la herramienta y ejecutarla. Esta es la manera en la que *Shiny* ordena sus elementos y es la manera en la que se ha configurado el controlador de la interfaz para poder acceder de manera ordenada a cada parte. La configuración se realiza mediante el elemento *"Advanced options and CPMs to use"* el cual despliega un nuevo menú (figura 3.4) con nuevas opciones el cual también es necesario poder manejar desde las pruebas, dando acceso a configuraciones avanzadas para los modelos, necesarias en flujos de trabajo complejos.
- En la figura 3.5: por último, la vista de resultados muestra una distribución también en tres bloques de los cuales el **bloque 2** no es interactuable ya que son figuras por lo que los datos que nos muestran no se pueden analizar de forma automática. Los datos se encuentran en el **bloque 3**, los cuales se pueden recoger para su posterior procesamiento mediante otras herramientas futuras. En esta vista también se ha cubierto la completa funcionalidad de la interfaz.

4

Implementación

En este capítulo se explica el proceso de creación e implementación de la arquitectura expuesta en el apartado 3.3, explicando las razones de la implementación, las herramientas empleadas y sus versiones necesarias sobre las que se ha desarrollado la totalidad de la herramienta. Se comentan también las dificultades encontradas y las soluciones propuestas para cada caso.

4.1. Creación y estructura del proyecto

Esta estructura la podemos ver en la figura 4.1:

```
EvAM-Tools-Web-tests
├── .vscode
│   └── launch.json
├── data
│   ├── BRCA_ba_s.csv
│   └── ov2.csv
├── src
│   ├── config.json
│   ├── evam_tools_web_tester.py
│   ├── test_case.py
│   ├── selenium_core
│   │   ├── __init__.py
│   │   ├── evam_tools_page_controller.py
│   │   └── selenium_wrapper.py
│   └── tests
│       ├── __init__.py
│       ├── test_example_1.py
│       ├── test_example_2.py
│       └── test_template.py
├── .env
├── .gitignore
├── requirements.txt
├── screenshots
│   └── screenshot.png
```

Figura 4.1: Estructura de directorios del proyecto

- En la raíz del directorio:
  - Fichero .env con la configuración básica y global de la herramienta.
- En el directorio 'src':
  - Punto de entrada y coordinado ("evam_tools_web_tester.py").
  - Fichero config.json el cual contiene la lista de pruebas divididas en sus respectivos grupos.
  - Fichero el cual contiene la clase Test Case genérica ("test_case.py").
  - Módulo Selenium Core ("selenium_core")
  - Módulo Tests ("tests")
- En el directorio 'data':
  - De momento se presentan dos ficheros como ejemplos para los tests que se han implementado junto con la herramienta.
- En el directorio 'screenshots' se contendrán las capturas de pantalla en caso de un error detectable en la ejecución de una prueba.

Por último, el archivo "requirements.txt" es donde se añaden referencias a los paquetes de "pypi.org" [23] que se requieren en el proyecto.

Como complemento se ha integrado la carpeta ".vscode" la cual contiene el fichero "launch.json" el cual nos permite lanzar las pruebas sobre el propio depurador de Visual Studio Code para facilitar su desarrollo.

4.2. Implementación de los módulos diseñados

4.2.1. Fichero .JSON (config.json)

Al igual que en el apartado de diseño, comenzamos describiendo la implementación del fichero de configuración.

La necesidad de poder crear grupos de una manera sencilla y ordenada se planteó de varias formas que, finalmente, implicaban la duplicación de código, poca expansibilidad del código sin afectar a otras pruebas y problemas relacionados con el entendimiento del código y la continuidad del desarrollo. Estos errores se solucionaron planteando un sistema por el que todas las pruebas se encontrarían en un fichero individual y su ejecución se planearía por medio de este fichero de configuración.

A continuación se muestra un ejemplo de configuración en la figura 4.2:

Figura 4.2: Fichero config.json

A la izquierda de la figura 4.2 observamos un fichero de ejemplo para tomar como referencia. En él se detalla la estructura que se sigue; el nombre del grupo será el primer elemento por el que se pueda filtrar sobre los datos, una vez obtenido el grupo que nos ocupa se obtiene la lista de todas las pruebas pertenecientes a dicho grupo. Cada una de las pruebas se puede configurar por separado con los argumentos facilitados como ejemplo, estos argumentos pueden ser ampliados de acuerdo con las necesidades futuras de la herramienta.

Se puede observar en la parte derecha de la figura 4.2 un ejemplo más completo, en el cual se describen dos grupos diferentes en el mismo fichero (”*intensive_tests*” y ”*simple_tests*”) y tres pruebas diferentes (”*test_example_1*”, ”*test_example_2*” y ”*test_example_3*”). Cabe destacar que el ”*test_example_1*” se encuentra presente en ambos grupos por lo que se ejecutará en ambas ejecuciones aunque con distintos argumentos.

4.2.2. Punto de entrada (evam_tools_web_tester.py)

Este fichero, como ya hemos comentado en el diseño en el apartado 3.3.2, se utiliza para la coordinación del resto de la herramienta. Los argumentos necesarios para la ejecución de la herramienta son el buscador sobre el que se va a realizar la prueba y el grupo de pruebas que ejecutará en esa iteración, el cual será utilizado para seleccionarlas del fichero ”config.json” apartado 4.2.1 el cual tras ser formateado con la librería *”JSON”* [24] de *Python*, es iterada prueba a prueba.

Antes de ejecutar la totalidad de las pruebas verificamos la configuración del buscador proporcionado creando una prueba de arranque y probando la conexión a la URL sobre la que se van a hacer las pruebas. En caso de fallar cualquiera de las comprobaciones la herramienta se detendrá.

Tras esta comprobación, la ejecución de las pruebas se realiza mediante un método ”*run_tests(tests_to_execute)*” el cual ha sido desarrollado para trabajar sobre el módulo ”tests” de la siguiente manera:

```
1   def run_test(tests_to_execute: dict) -> None:
2
3       test_name = tests_to_execute["module"]
4       args = tests_to_execute["args"]
5       args["name"] = test_name
6
7       try:
8           test_module = __import__("tests." + test_name)
9       except ModuleNotFoundError:
10          print(
11              f"Test module [{test_name}] not found. Make sure this test is located in the tests
                    folder"
12          )
13          return
14
15      test_class = getattr(test_module, test_name)
16      test_instance = test_class.TestMethod(**args)
17      print("Running test: " + str(test_name))
18      test_instance.run()
```

Figura 4.3: Función para la configuración y ejecución de pruebas.

El correcto funcionamiento de la herramienta depende de la implementación de las pruebas en el paquete ”tests” discutido más adelante. Esto se debe a que, como hemos descrito antes, esta función de ejecución trabaja directamente sobre el paquete haciendo referencia a nombres de clases y métodos específicos de la siguiente manera.

Para conseguir esa modularidad que se busca, el método ”*__import__*” (línea 8 de la figura 4.3) es utilizado para obtener el código presente en otro fichero mediante su nombre; sacado anteriormente de el fichero .json. A continuación la función ”*getattr*” (línea 15 de la figura 4.3) nos permite crear un objeto de la clase importada para su uso.

4.2.3. Tests (/src/tests/)

Para la implementación del paquete ”test” se plantearon varias opciones entre las que se encontraban el uso de clases compatibles para ser utilizadas con paquetes ya disponibles de pruebas para *Python* como ”*unittest*” [25] el cual comenzó siendo una propuesta atractiva debido a ser un framework completo de pruebas con bastante trayectoria y fiabilidad. No obstante los inconvenientes no tardaron en aparecer en cuanto aplicamos los requisitos enfocados a la reutilización de tests, creación de grupos y fácil implementación sin duplicación masiva de código, dejando esta funcionalidad reservada para la prueba de la herramienta en lugar del uso en la misma.

Para conseguir los objetivos descritos la estructura del paquete ha de ser la más simple posible, optando finalmente por la estructura mostrada en la figura figura 4.4:

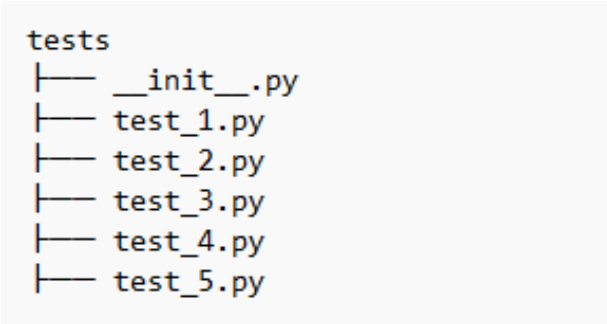

Figura 4.4: Estructura del paquete de ”tests”

Para poder indicarle a *Python* que una carpeta sea considerada un paquete, debe contener un archivo de inicio *”__init__.py”* el cual puede estar completamente vacío pero es necesario.

Haciendo referencia a la dependencia descrita en el aparatado anterior apartado 4.2.2, la clase sobre la que **se deberán** construir futuras pruebas, el esqueleto principal, se encuentra en el fichero ”*test_template.py*”, dentro del paquete *”tests”*; un módulo del paquete que no ejecuta ninguna prueba sobre la interfaz pero tiene la estructura mínima para funcionar con el resto de la herramienta. Las creación de nuevas pruebas basarán su contenido en esta.

La propia clase indica la obligación de nombrar a la clase que contiene la prueba a ejecutar con el nombre ”*TestMethod*”, nombre de clase que ya se ha referenciado en la figura 4.3, al igual que la obligación de tener un método ”*test_body*” en el cual se implementará el flujo de trabajo principal que seguirá la prueba, esta estructura permite tener más funcionalidad para acompañar a cada prueba como ya hemos indicado en el diseño, gracias a esto se pueden implementar ejecuciones no necesariamente orientadas a pruebas de funcionalidad, si no también como automatizaciones de ejecuciones que sean repetitivas y costosas de ejecutar varias veces, permitiendo en el procesamiento de los datos en el propio módulo. La estructura descrita de la prueba la podemos observar en la figura 4.5

```
from test_case import TestCase

class TestMethod(TestCase):
    def __init__(self, **config):
        super().__init__(**config)

    def test_body(self):
        print("Test template")
```

Figura 4.5: Plantilla para pruebas

4.2.4. Test case (test_case.py)

La estructura descrita en el apartado anterior hereda de una clase que contiene la funcionalidad principal necesaria para hacer funcionar el test, siendo el nexo de unión entre el controlador tanto de Selenium como el controlador de la interfaz. Esta clase, denominada "*TestCase*" es la clase que contiene los argumentos predeterminados los cuales pueden ser configurados desde el fichero "*config.json*" apartado 4.2.1. La clase contiene tres funciones desarrolladas las cuales tienen funcionalidades básicas para el control del buscador. Estas funciones también están estandarizadas para todos los tests pudiendo personalizarlas en cada una de las pruebas. Al igual que sucede en la implementación de nuevas pruebas se debe seguir un cierto criterio para armonizar el funcionamiento de la herramienta. Como ocurre en apartado 4.2.2, la función busca por un método "*test_body*" el cual es **obligatorio**.

```
def run(self) -> None:
    self.set_up()

    try:
        method = getattr(self, "test_body")
        method()
    except Exception as e:
        print(f"Error while running test: {e}\n")

        if self.large_log:
            print(traceback.format_exc())

        if self.screenshots:
            self.wrapped_driver.save_screenshot("./screenshots/" + str(self.name) + "_error.png")

    self.tear_down()
```

Figura 4.6: Función 'run' de la clase test_case

4.2.5. Selenium core (/src/selenium_core/)

Para la implementación de este paquete, al igual que sucede con el paquete *"test"*, se incluye *"__init__.py"* para poder ser llamado desde cualquier prueba. En los apartados: apartado 4.2.5 y apartado 4.2.5; se describe la funcionalidad prestada por ambos módulos que, resultan cruciales en el manejo de la interfaz web debido a que Selenium es un framework enfocado a satisfacer las necesidades de una gran cantidad de casos, cuenta con funciones generales y con poca especificación. Esto hace que existan casos en los cuales la precisión de la herramienta no es suficiente y se necesita de desarrollos más profundos.

• Selenium wrapper (selenium_wrapper.py)

- Funciónes de conexión para navegadores

Dado que la conexión mediante un navegador con la aplicación web es una acción que será repetida en cada prueba, la encapsulación de la misma es necesaria para mantener una experiencia fluida para su uso. Estas funciones son personalizadas para cada tipo de buscador compatible, haciendo caso a las peculiaridades y configuraciones de cada uno. Estas funciones son las encargadas de configurar el navegador con los argumentos necesarios para cada una de las pruebas, procedentes de la lectura del fichero *"config.json"* (apartado 4.2.1) y realizar la conexión con la URL de la aplicación web establecida en el fichero *".env"*.

Una vez creadas estas funciones, cualquier argumento de configuración sobre el navegador será añadido en ellas sin afectar a los códigos presentes en las pruebas ya que, como se puede observar en la figura 4.6 en la línea 2, las pruebas llaman al comienzo de sus ejecuciones a la función "*set_up()*" la cual se encarga, de forma transparente, de llamar a esta configuración.

- Funciones de espera para la carga y descarga de elementos

Las funciones de la figura 4.7 implementan funciones de espera para comprobar la presencia y visibilidad, como la desaparición de elementos en la interfaz web. Originalmente estas funciones no presentaban esta funcionalidad ya que en caso de no cumplir el requisito de la función en un tiempo determinado por un argumento '*timeout*', se lanzaba una excepción terminando por completo el programa. Esta funcionalidad resulta interesante pero también se presentan circunstancias en las que el comportamiento de esta determine el camino a seguir más adelante, cosa que, no se podía lograr de manera nativa en Selenium. Esta condición puede devolver un error por varios motivos, como no encontrar el elemento en la interfaz o terminar el tiempo de espera, por lo que controlar el comportamiento con una excepción genérica es una mejor opción puesto que el comportamiento será idéntico en todos los casos.

```
1   def wait_invisibility_by_xpath(self, xpath, timeout=10) -> bool:
2       try:
3           WebDriverWait(self.driver, timeout).until(
4               ec.invisibility_of_element_located((By.XPATH, xpath))
5           )
6           return True
7       except Exception:
8           return False
9
10  def wait_visibility_by_xpath(self, xpath, timeout=10) -> bool:
11      try:
12          WebDriverWait(self.driver, timeout).until(
13              ec.visibility_of_element_located((By.XPATH, xpath))
14          )
15          return True
16      except Exception:
17          return False
```

Figura 4.7: Funciónes de espera

- Función de localización por texto

```
1   def find_element_by_text(self, tag_name, text, ui_element=None) -> None:
2       if ui_element is None:
3           ui_element = self.active_tab
4
5       for elem in ui_element.find_elements_by_tag_name(tag_name):
6           if elem.text == text:
7               return elem
8
9       return None
```

Figura 4.8: Función de localización por texto

Selenium proporciona funciones de localización basadas en la estructura de todos los elementos que se pueden encontrar en un documento HTML. No obstante, para aplicaciones web auto-generadas, como es el caso, ciertos elementos comparten la misma estructura haciendo imposible distinguirlos de otra manera que no sea mediante el texto que muestran en la interfaz. Debido a esto surgió la necesidad de desarrollar un localizador por texto (figura 4.8) que funciona haciendo uso de la función "*find_element_by_tag_name*" perteneciente de forma nativa en Selenium, y encapsulando su uso entorno a la identificación del texto de un elemento perteneciente a una determinada clase. Por lo que no es necesario facilitar solo el texto del elemento sino también la clase de elemento HTML en el que se encuentra. Podemos observar un ejemplo en el que es necesario esta funcionalidad en la figura 4.9 donde dos elementos comparten la misma estructura para crear un botón y siendo lo único que los distingue el texto de cada uno.

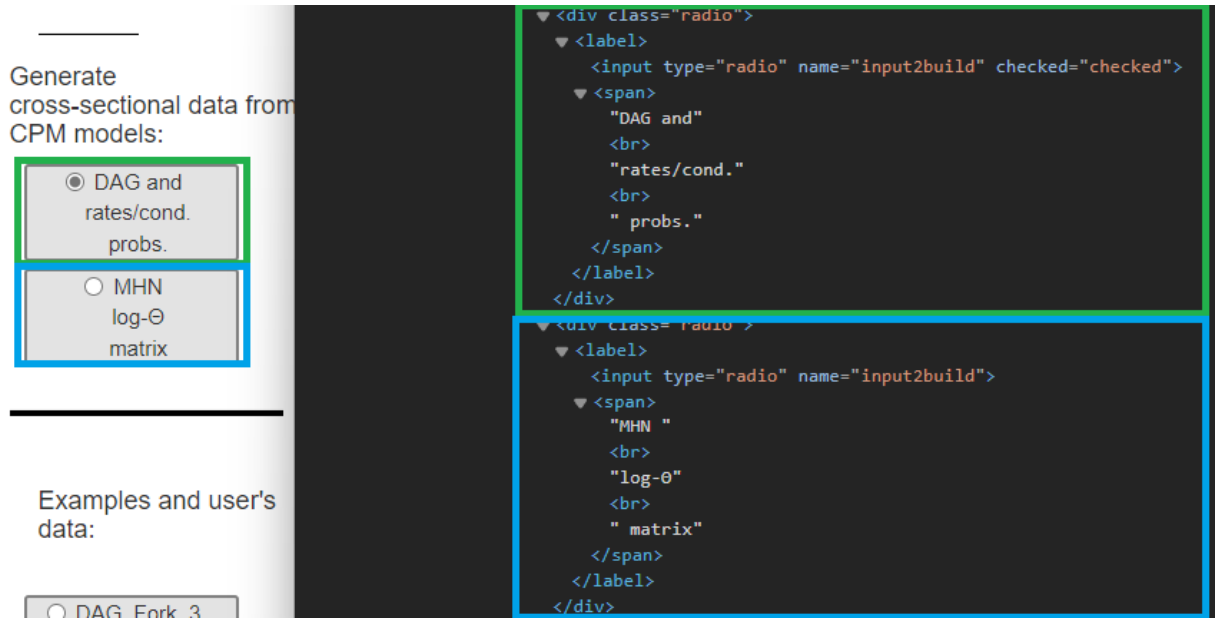

Figura 4.9: Ejemplo de elemento duplicado de la interfaz de EvAM-Tools

- Función para traer elementos a la vista del navegador

```
1    def scroll_into_view(self, element):
2
3        self.driver.execute_script("arguments[0].scrollIntoView();", element)
4        self.driver.execute_script("window.scrollBy(0,-50)")
5        sleep(1)
6
7        return element
```

Figura 4.10: Función para tarer elemento a la vista en un buscador

Por último, y para terminar con las funciones que aplican al control del propio buscador, la posibilidad de desplazarse de manera vertical en el buscador no se puede realizar de manera nativa desde *Selenium* por lo que es imperativo la creación de la funcionalidad para casos como los ajustes avanzados vistos en figura 3.4. Aquí no es posible hacer visibles todas las opciones en pantalla, esta función podría variar en cada buscadores más exóticos, no obstante es compatible tanto fon *Firefox* como con *Chrome*.

• EvAM Tools Driver (evam_tools_driver.py)

Habiendo descrito la parte encargada de tomar el control sobre el buscador de manera que satisfaga nuestras necesidades toca hacer lo mismo por parte del controlador de la interfaz web. Como se ha descrito antes, la interacción con los elementos de una interfaz web generada por medio de algún tipo de automatización puede resultar un problema para *Selenium* porque las estructuras tienden a

repetirse o ser muy complejas para ser manejadas por comandos simples como pulsar sobre ellas. Para solucionar este problema se desarrolla el controlador de página, una batería de funciones personalizadas e implementadas específicamente para los problemas presentes en la interfaz que nos ocupa.

El desarrollo, como ya se ha introducido anteriormente, se basa en la adaptación y especificación de las funciones más básicas de *Selenium* sobre elementos complejos para facilitar su reutilización, ya que una vez encapsulada la funcionalidad de un elemento se puede reutilizar para todos los similares existentes o futuros elementos que se añadirán en la aplicación. Adicionalmente a las estructuras más complejas, las acciones más comunes presentes en la mayoría de las pruebas también han sido encapsuladas para su uso más sencillo y controlar posibles errores que puedan darse durante la ejecución de los mismos.

Para poder seleccionar de manera más específica, en la mayoría las funciones descritas a continuación el elemento "*ui_element*" aparece referenciado y utilizado en ellas. Se trata de una manera de acotar el alcance de la identificación de elementos, evitando así seleccionar por error o interaccionar con un elemento duplicado en la vista que no sea el deseado. Este elemento puede ser seleccionado empleando las funciones básicas de localización otorgadas por selenium. utilizando el elemento seleccionable más pequeño que encapsule el elemento con el que se quiera interactuar finalmente.

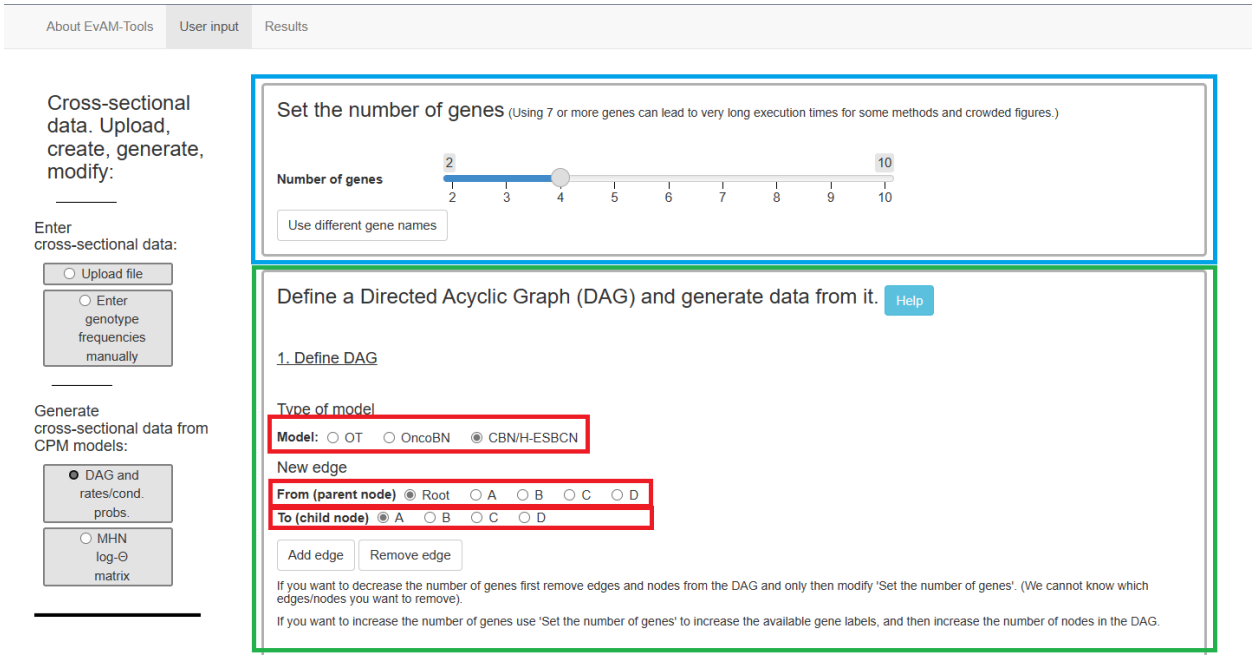

Figura 4.11: Ejemplo de elementos anidados para selección de "ui_element"

Como podemos observar en la figura 4.11, en caso de querer interactuar con la barra deslizante nos conviene seleccionar el elemento resaltado con el recuadro azul, correspondiendo a un 'div' de HTML, o cualquier otra estructura que pueda contener otras, que contiene unicamente esa barra, acotando la acción sobre ese elemento. En cambio, si quisiéramos interactuar sobre la lista indicada con el texto

"*From (parent node)*" podríamos seleccionar el elemento resaltado con el cuadro verde, pero esto haría que, dependiendo del control implementado sobre la prueba en cuestión, se pueda estar afectando a otros elementos que comparten estructura con el que nos interesa. Por tanto, es más interesante seleccionar como elemento de la interfaz el rodeado por un cuadro rojo el cual contiene la lista de viñetas con la que queremos realizar una acción.

Como hemos descrito en el apartado 2.1.1, la funcionalidad proporcionada por EvAM-Tools como aplicación científica es muy amplia, por eso necesitamos cubrir los casos de uso para toda ella. Esto se ha conseguido implementando pruebas sobre todos los casos de uso disponibles en los ejemplos de su documentación [8] y para facilitar la implementación de futuras pruebas se han diseñado e implementado una serie de funciones personalizadas, utilizando las proporcionadas por Selenium, que se adaptan por completo a los elementos presentes en todas las vistas. Estas funciones solucionan problemas de control sobre elementos complejos que solo se podían probar de manera manual ya que, las herramientas de automatización no eran capaces de recrear su comportamiento

- **Funcionalidad común entre vistas**:

  - **navbar_to(self, tab_name, timeout)**: Función encargada de cambiar de vista entre las disponibles.

- **Funcionalidad presente en la vista 'User input'**:

  - **select_from_cross_sectional_data(self, tag_name)**: Como se discutió en el diseño, cada vista de la interfaz está. dividida en varias partes, esta función permite acceder a todos los elementos de la vista de 'User input' del bloque 1 figura 3.3.

  - **run_evamtools(self, timeout)**: Función para ejecutar el análisis de los datos.

  - **toggle_advanced_options_on(self,)**: Abrir menú de opciones avanzadas.

  - **toggle_advanced_options_off(self,)**: Cerrar menú de opciones avanzadas.

  - **set_advanced_options(self, option_name, option_value, timeout)**: Configurar una opción avanzada. Desde esta función pueden ser editadas todas las opciones disponibles.

  - **change_genotypes_count(self, index=[], counts=[])**: Cambiar el número de genotipos, una opción disponible en todas las vistas.

  - **rename_data(self, data_to_rename, rename)**: Cambiar el nombre del grupo de datos seleccionados.

  - **upload_data_set(self, name, file_path: str)**: Subir datos desde un fichero en la vista 'Upload file'.

  - **set_number_of_genes(self, number_of_genes: int)**: Insertar numero de genes desde la vista 'Enter genotype frequencies manually'

  - **add_genotype(self, mutations: list, count: int)**: Insertar nuevo gen desde la vista 'Enter genotype frequencies manually'

  - **define_DAG(self, model, parent_node, child_mode, action)**: Definir el modelo DAG des-

de la vista 'DAG and rates/cond. probs'.

- **define_DAG_table(self, from_to, relation, lambdas)**: Definir la tabla del modelo DAG desde vista 'DAG and rates/cond. probs'.

- **generate_data_from_DAG_model(self, epos, num_genotypes, observation_noise)**: Inserción del resto de datos para el modelo DAG.

- **define_MHN(self, values)**: Definir la matriz de datos para el modelo MHN.

- **generate_data_from_MHN_model(self, num_genotypes, observation_noise)**: Inserción del resto de datos para el modelo MHN.

- **Funcionalidad presente en la vista 'Results'**:

  - **CPMs_to_show(self, check: list)**: Selección de CPMs visibles.

  - **predictions_from_fitted_models(self, option)**: Selecciona el modelo de predicciones.

  - **type_of_label(self, option)**: Selecciona el tipo de etiqueta.

  - **relevant_paths_to_show(self, value)**: Selecciona el número de caminos relevantes.

  - **download_results(self)**: Descarga los resultados.

Estas funciones solucionan problemas arrastrados por la actualización dinámica de la página donde por ejemplo; la sección encargada de modificar un modelo DAG actualiza el número de genes disponibles para la definición del modelo, en base al elemento encargado de seleccionar dicho número, esto recarga los elementos de manera dinámica, variando en cada flujo dependiendo de los datos de entrada como podemos observar en la figura 4.12.

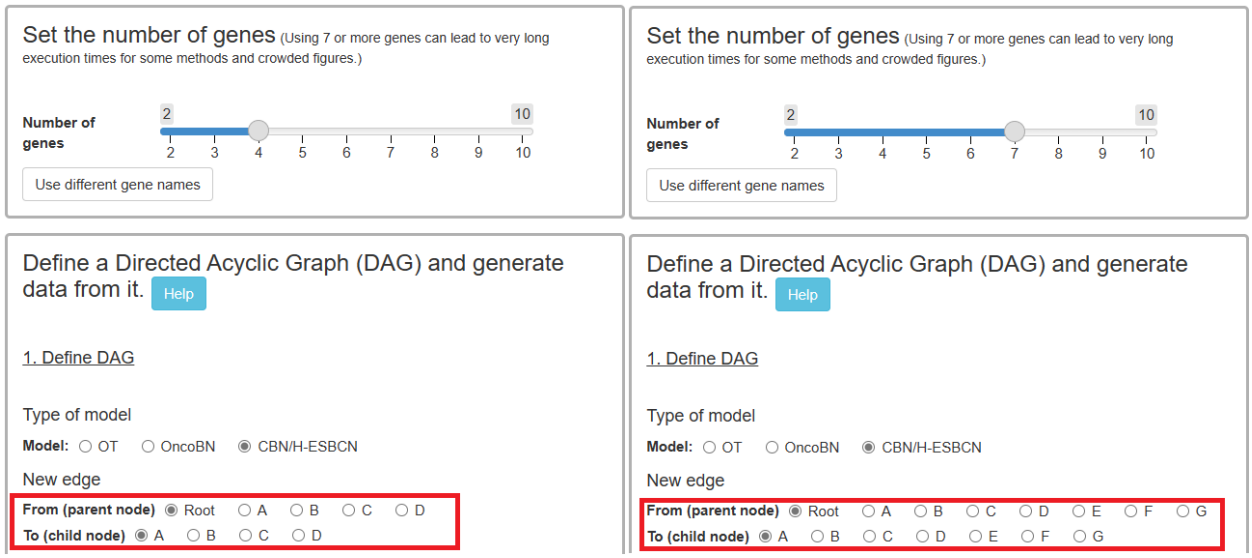

Figura 4.12: Ejemplo de actualización dinámica de elementos en la interfaz. Modelo DAG

Otro caso donde podemos ver un comportamiento similar se da en la modificación del modelo MHN, donde el mismo elemento de selección de número de genes varía la forma de la tabla como podemos observar en la figura 4.13.

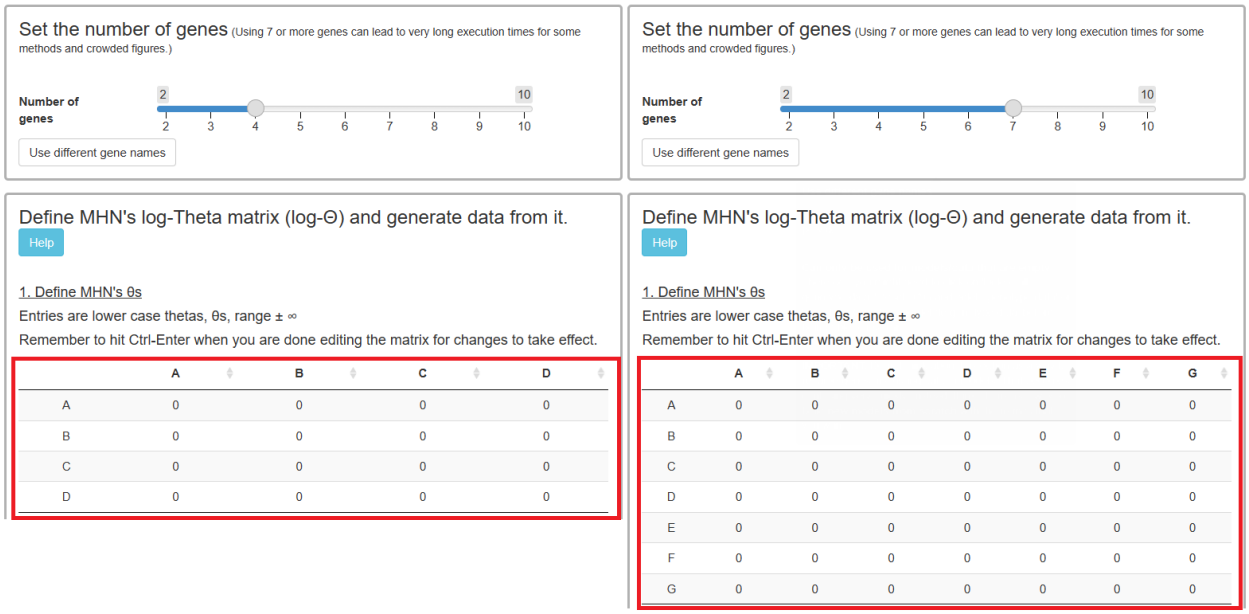

Figura 4.13: Ejemplo de actualización dinámica de elementos en la interfaz. Modelo MHN

Toda esta actualización es manejada gracias a la implementación de funcionalidad capaz de identificar y manejar estas estructuras, pudiendo interactuar sobre ellas de manera dinámica sin estar sujetos a unos parámetros fijos. Esto permite evitar la necesidad de modificar el código del controlador a la hora de implementar nuevas pruebas con valores diferentes o en caso de cambiar los límites permitidos.

Algunas de las funcionalidades anteriores aplican su funcionalidad sobre elementos de la interfaz presentes en otras secciones de la web, para evitar la duplicación de código, dicha funcionalidad de elementos es encapsulada para soportar la actualización dinámica a lo largo de todos los casos de uso de la web. Estas funciones elaboradas sobre las disponibles en Selenium, facilitan el uso de la herramienta para implementar pruebas. Su desarrollo se ha diseñado entorno a la flexibilidad y resolución de problemas que acarrean las funcionalidades dependientes de otros elementos como se ha discutido anteriormente, cubriendo todos los casos de uso donde la actualización dinámica de contenido resulta ser un problema para la automatización de acciones sobre la interfaz. Todas las funciones necesarias para el manejo de los elementos presentes actualmente en la aplicación web de EvAM-Tools se describen a continuación.

- Función para seleccionar datos de una lista de verificación

Las listas de verificación están presentes en todas las vistas y el control de su uso resulta crítico para el correcto avance en el flujo de trabajo. Estos elementos son los más susceptibles a cambios por lo que se ha buscado una implementación genérica no basada en los elementos existentes actualmente ni en un número fijo. De esta forma, en caso de eliminar un elemento de una lista en un futuro, las pruebas anteriores que implementaran esta función no se verían afectadas en absoluto (figura 4.14).

```
1        def select_from_checklist(self, ui_element, check: list) -> None:
2            self.scroll_into_view(ui_element)
3
4            for option in ui_element.find_elements_by_xpath("./div"):
5                option_element = option.find_element_by_xpath("./label")
6                if option_element.text in check and not option_element.find_element_by_xpath("./input")
                     .is_selected():
7                    option_element.click()
8                elif option_element.text not in check and option_element.find_element_by_xpath("./input
                     ").is_selected():
9                    option_element.click()
```

Figura 4.14: Función para seleccionar datos de una lista de verificación

- Función para seleccionar datos de una lista desplegable

Al igual que las listas de verificación, la selección desde una lista desplegable presenta una funcionalidad parecida, registrando todos los elementos del desplegable en busca del indicado como argumento. En caso de no aparecer en la lista simplemente no se seleccionará ningún elemento, haciendo que esta función también sea tolerante a cambios en las opciones suministradas por la aplicación (figura 4.15).

```
1        def select_from_dropdown(self, ui_element, option) -> None:
2            self.scroll_into_view(ui_element)
3            ui_element.find_element_by_class_name("selectize-input").click()
4            option_element = ui_element.find_element_by_xpath(f".//div[@class='option'␣and␣text()='{
                 option}']")
5            option_element.click()
```

Figura 4.15: Función para seleccionar datos de una lista desplegable

- Función para seleccionar de una barra deslizante acotada

La funcionalidad de una barra de selección resulta un tanto más compleja ya que Selenium no está preparado de ninguna forma para poder actuar sobre este tipo de elementos de manera nativa con una sola función, y hace falta el uso del elemento 'ActionChains' [26] para encadenar acciones

de manera automática. Este módulo nos permite ejecutar acciones de manera secuencial en una sola instrucción. Como funcionalidad adicional, la función implementada tiene en cuenta los rangos entre los que se pueden seleccionar valores de la barra por lo que no es posible forzar valores que puedan comprometer la funcionalidad de la aplicación web por un uso indebido de la herramienta. El manejo de las barras deslizantes se ha estandarizado para todos los tipos de barra existentes en la interfaz actualmente, ya que algunas presentan facilidades no presentes en las demás, habiendo estandarizado el método más complejo, el cual es efectivo en todas las barras sin excepción.

```
1        def select_from_sliderbar(self, ui_element, value) -> None:
2            self.scroll_into_view(ui_element)
3
4            slider_min_value = int(ui_element.find_element_by_class_name("irs-min").text)
5            slider_max_value = int(ui_element.find_element_by_class_name("irs-max").text)
6
7            if value < slider_min_value or value > slider_max_value:
8                raise ValueError(f"Value must be between {slider_min_value} and {slider_max_value}")
9
10           slider_value = int(ui_element.find_element_by_class_name("irs-single").get_attribute("
                 textContent"))
11
12           slider = ui_element.find_element_by_class_name("irs-handle")
13           direction = Keys.ARROW_RIGHT if value > int(slider_value) else Keys.ARROW_LEFT
14
15           while slider_value != value:
16               ActionChains(self.driver).click_and_hold(slider).send_keys(
17                   direction
18               ).release().perform()
19               slider_value = int(
20                   ui_element.find_element_by_class_name("irs-single").get_attribute("textContent")
21               )
```

Figura 4.16: Función para seleccionar de una barra deslizante acotada

- Función para seleccionar de una lista de viñetas

El último método utilizado para la selección de datos, la lista de viñetas presenta un comportamiento similar a la la lista desplegable haciendo que su implementación se base en el recorrido de opciones disponibles en el elemento de la interfaz seleccionado, finalizando en caso de encontrarlo, saliendo en caso de no encontrarlo.

```
1        def select_from_bullet_list(self, ui_element, option) -> None:
2            self.scroll_into_view(ui_element)
3            option_element = self.find_element_by_text('span', option, ui_element)ui_element.
                 find_element_by_xpath(f".//span[text()='{option}']")
4            option_element.find_element_by_xpath("./preceding-sibling::input").click()
```

Figura 4.17: Función para seleccionar de una lista de viñetas

- Función para carga de contenido

Tratándose de una página dinámica generada en función de las interacciones con sus elementos, el desfase en la respuesta puede ser un inconveniente con la interacción de sus elementos. Esto se debe a que los elementos no se cargan por completo a la vez, si no que lo hacen de manera secuencial, haciendo un uso excesivo de las funciones de espera. Para evitar esto, la función mostrada en la figura 4.18 espera a la carga de todos los elementos de la vista activa para realizar la siguiente acción. Excluyendo los gráficos en caso de estar en modo sin interfaz ya que no aportan información útil y ralentizan la ejecución de las pruebas.

```
1         def load_content(self, div=None) -> None:
2             if div == None:
3                 div = self.active_tab
4             for elem in div.find_elements_by_xpath("./*"):
5                 self.wait_visibility_by_xpath(elem, 10)
6                 if elem.tag_name == "div" and "plot" not in elem.get_attribute("id"):
7                     if self.headless and "plot" in elem.get_attribute("id"):
8                         continue
9                     self.load_content(elem)
```

Figura 4.18: Función para carga de contenido

5

Pruebas

Debido a que la idea de la herramienta es concienciar y agilizar el proceso de pruebas de un desarrollo, las realizadas sobre la misma han sido igual de exhaustivas y completas como se espera que se realicen con ella. Las pruebas que se han ido realizado han variado en función de la etapa del desarrollo en la que nos encontrábamos, teniendo pruebas más unitarias en torno al principio del mismo para evitar propagar errores y comenzar sobre una base sólida. Como se ha podido observar, la conexión entre módulos y la correcta sincronización de los mismos es una parte crucial para el funcionamiento, por lo que se ha probado de manera exhaustiva que estos enlaces sean sólidos ante cualquier eventualidad no planeada. Por último, cuando la herramienta se acercaba a su estado final se fueron probando diferentes entornos y configuraciones para saber hasta qué punto se podría forzar la flexibilidad de la misma y cuales son sus limitaciones a partir de las cuales la funcionalidad prestada peligra.

Como se ha indicado, las primeras pruebas fueron pruebas unitarias sobre los primeros módulos desarrollados. Cuando se planteó el diseño de la herramienta se comenzaron programando las funciones encargadas del control sobre el buscador y la conexión con la aplicación; las pruebas sobre este elemento fueron sencillas y las comprobaciones de las mismas acabaron formando parte de las funciones de validación del entorno, ya que al poder pasar esas pruebas nos aseguramos de una ejecución satisfactorios por parte de este módulo. La funcionalidad probada ha sido la mínima necesaria:

- Ejecución en modo 'headless' para no abrir una interfaz al realizar las pruebas.
- Maximización de pantalla.
- Uso de ventanas privadas.
- Conexión sin espera de certificados.

Con esta funcionalidad probada fue más que suficiente para validar el entorno del buscador. No obstante más funcionalidad está disponible.

Una vez cubierta la funcionalidad básica para crear una conexión, la interacción con la interfaz web pasó a ser el principal objetivo del desarrollo, siendo esta la parte que más pruebas necesita para asegurar su funcionamiento. Cada una de las funciones descritas en la implementación del controlador

de la página apartado 4.2.5 ha seguido la misma ruta de pruebas unitarias, siendo necesario probar más en profundidad debido a la complejidad de los casos contemplados, la ruta seguida para la prueba de todas estas funciones se ha basado en:

- Correcta localización del elemento dentro de la interfaz web.
- Correcta interacción básica por parte de selenium, entendiendo por iteración básica los siguientes casos:
    - Carga completa al realizar la carga del contenido de la pestaña activa.
    - Visibilidad.
    - Respuesta a clicks.
    - Recogida de texto presente en el elemento.
- Comprobación de datos de salida conocidos.
- Comprobación de datos de salida desconocidos.
- Introducción de datos de entrada válidos.
- Introducción de datos de entrada no válidos.
- Control de excepciones.
- Interacciones sobre elementos no encontrados en la interfaz.

Aunque las pruebas básicas por las que pasaban todos los elementos eran suficiente para solventar la mayoría de los problemas presentes, ciertas estructuras podían pasar todos estos casos de una manera insatisfactoria sin levantar ninguna alarma, por lo que varias funciones necesitaron un cuidado especial a la hora de validar su funcionamiento. Este es el caso de la función encargada de mover una barra deslizante, la cual funciona perfectamente si se interacciona por un usuario desde la interfaz, pero al presentar una estructura dinámica con la resolución de la pantalla necesitábamos comprobar no solo que la función se moviera en la dirección que debía, el número de veces que debía hacerlo, si no que realmente realizara la función de mover el indicador a la cantidad deseada en las pruebas automatizadas.

Otra función que necesitó un trato diferente para validarse fue la encargada de cargar el contenido de la ventana actual, debido a que estas pruebas se pueden realizar sobre la aplicación final sobre la que trabaja directamente el usuario. Esto puede generar tiempos de carga entre elementos y ya que hay ciertos elementos que se necesitan para la localización de sus elementos hijos que, en caso de no cargarse por completo no permiten la interacción total por parte de la herramienta, acaban creando errores debido a la falta de elementos cargados.

Esta sección de pruebas fue las más costosa en cuanto a tiempo empleado en ella pero finalmente todas las funciones necesarias para la interacción con elementos de la interfaz han sido probados y se ha asegurado un correcto funcionamiento de todos ellos.

No obstante, esta no fue la última parte que necesitaba ser probada. Todas estas funciones deben ser alojadas en un fichero de pruebas el cual debe ser capaz de ser ejecutado por el punto de entrada de datos y comunicarse con el buscador para realizar las acciones.

La prueba de la estructura de la aplicación ha sido lo último en ser probado, una vez sabíamos que los cimientos eran sólidos y que todas las partes por separado funcionaban se necesitaba saber que el conjunto se organizaba correctamente. Para realizar estas pruebas se crearon varios ficheros de pruebas reales con errores de estructura conocidos, los errores probados fueron:

- Generar una prueba sin el nombre necesario por defecto.
- Generar una prueba sin la función 'test_body' necesaria para ejecutar las pruebas.
- Configurar en el fichero 'config.json' nombres de pruebas que no existen.
- Configurar un buscador sin controlador disponible.
- Ejecutar las pruebas sobre una URL no compatible con el controlador de la página.

Por último se realizaron pruebas de rendimiento para asegurar que la aplicación presenta una utilidad real y no consume más tiempo que un usuario real en ejecutar las tareas asignadas.

Una vez probada la herramienta y su correcto funcionamiento, se realizaron las pruebas de todos los ejemplos de uso descritos en la documentación de ejemplo [8]. Estas pruebas son las encargadas de validar el correcto funcionamiento de la web, siendo las que se deban pasar siempre que se realice un nuevo cambio en la aplicación en R que afecte la funcionalidad de la aplicación. Todas estas pruebas están a disposición de los desarrolladores futuros para su uso en el proyecto final.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

El principal objetivo de este TFG era implementar una herramienta que permita la prueba y validación de casos de uso de una manera sencilla y cómoda para el desarrollador sobre una interfaz web dinámica generada con Shiny. Tras el diseño personalizado de la herramienta adaptado a la interfaz, la implementación y las pruebas sobre la herramienta, se considera haber cumplido con el objetivo planteado.

La herramienta finalizada resuelve todos los problemas e inconvenientes presentes en otras aplicaciones del estilo como pueden ser la falta de integración con el entorno de desarrollo para la aplicación principal, la falta de control sobre los elementos de la interfaz dinámica, entre otros.

La utilización de esta herramienta ha permitido automatizar, simplificar el proceso de pruebas en la interfaz lo que ha llevado a una mayor eficiencia en la ejecución de pruebas, ahorro de tiempo y esfuerzo en comparación con las pruebas manuales. Así mismo ha conseguido corregir errores y fallos de manera más rápida y precisa.

El enfoque modular y orientado a la expansión futura de los componentes de la herramienta ha demostrado ser beneficioso. Esto facilitará su mantenimiento y escalabilidad así como, gracias a las funciones creadas con la filosofía del cambio, la tolerancia a cambios sobre los elementos actuales de la interfaz web, permitiendo a las pruebas de validación generales de funcionamiento de elementos seguir siendo fiables incluso al eliminar o agregar elementos ya modelados por la herramienta, siendo mínima la intervención necesaria sobre las pruebas existentes llegando en algunos casos a no ser necesaria por completo.

Hablando de las pruebas sobre elementos, otro objetivo que se da por cumplido ha sido garantizar la cobertura sobre todos los elementos y funcionalidad actual de la interfaz y aplicación web, habiendo generado pruebas de validación de funcionamiento los cuales pueden utilizarse a partir de la finalización de este trabajo para desarrollos futuros asegurando así, con cualquier nueva funcionalidad agregada, no afectar al correcto funcionamiento anterior.

La expansión de estas pruebas, se puede hacer de manera sencilla simplemente añadiéndose nuevos elementos al módulo ”Tests” apartado 3.3.3 utilizando la plantilla generada y partiendo de las pruebas ya programadas, las cuales contienen ejemplos de uso de todos los elementos lo cual se puede tomar como referencia. Esta manera de expansión minimiza el esfuerzo invertido en la creación de pruebas y facilita la ejecución de las mismas.

Esto nos permite también mantener una batería de tests bastante grande sin un impacto en la complejidad del sistema global.

Cabe destacar que no se conocía en profundidad el alcance de Selenium ni las estrategias de pruebas que se necesitan seguir a la hora de garantizar la funcionalidad de una interfaz web. Por eso este proyecto ha sido un proceso de aprendizaje de lo que es una de las herramientas más potentes y valoradas para aplicaciones web que se tienen en la actualidad y el descubrimiento de la importancia y los beneficios de la inversión de tiempo en una de las etapas más importantes en el desarrollo de un producto software, que es la fase de pruebas y validación del trabajo desarrollado.

Actualmente el código fuente de la herramienta se encuentra en un repositorio público a disposición de todos los desarrolladores: `https://github.com/rdiaz02/EvAM-Tools-Web-tests`.

6.2. Trabajo futuro

La herramienta en su estado actual es completamente funcional y cumple con todos los requisitos descritos y necesarios. Como ya se ha discutido en el apartado 3.1, la herramienta es compatible con los buscadores más populares en la actualidad, Firefox y buscadores basados en Chromium. No obstante es posible que en un futuro se necesite ampliar esta funcionalidad a medida que los estándares cambien.

También, aunque la información recolectada durante la ejecución de las pruebas es suficiente para localizar errores, puede ser interesante implementar más medidas de recolección de datos. Uno de los más interesantes planteados es mejorar la captura de pantalla, escalando las capturas de pantalla por una grabación completa de la ejecución y guardando la grabación en caso de error, desechándose en caso de éxito pudiendo, mediante la configuración del fichero ”config.json” elegir si la grabación se guardará independientemente de la salida.

Continuando con la idea de recoger más datos en las ejecuciones, se puede plantear el uso de la herramienta, a parte de como herramienta para pruebas y validaciones de cambios, como ejecutador automático de casos de uso específicos y recurrentes que necesiten de varias ejecuciones cambiando datos o valores de entrada en cada iteración y guardando los datos en un informe o un formato de datos que pueda ser utilizado en otra aplicación. Una de las funcionalidades comenzadas a desarrollar, relacionada con la idea anterior ha sido recoger los datos presentes en las tablas de salida de la vista

de resultados para su posterior procesamiento.

De cara a mantener la facilidad en el uso de la herramienta en el futuro, un desarrollo interesante planteado pero no priorizado actualmente es el desarrollo de una imagen de Docker con una batería de pruebas por defecto para probar distintas funcionalidades, con una configuración de buscador y entorno ya predeterminada y asegurando un funcionamiento óptimo. Esta imagen sería pública lo que permite, corriendo un simple comando de Docker, levantar un contenedor, ejecutar las pruebas, recibir un resultado y no necesitar interactuar con la herramienta de ninguna manera si lo que se necesita es una validación rápida sin entrar en detalle. Junto con esta idea se esbozó también, la creación de una interfaz simple para el usuario de tal manera que la interacción con el código se resumiera simplemente a la elaboración de las pruebas, haciendo el proceso de ejecución y validación con una interfaz algo más amigable, intuitiva y sencilla.

Por último, dado que los desarrollos actuales se realizan sobre máquinas en Linux, la migración a otros sistemas no se ha planteado como algo prioritario. No obstante, incluso siendo Linux el sistema elegido por la mayoría de desarrolladores, el desarrollo en sistemas como Windows es una realidad y permitir la ejecución de la herramienta en dicho sistema sin errores y contemplando todas las casuísticas presentables es un punto a tener en cuenta para facilitar el acceso al desarrollo futuro de la aplicación al máximo número de usuarios posibles. La intención tras terminar este TFG no es dar la herramienta por terminada sino implementar parte de las mejoras discutidas junto con las que se descubran en el desarrollo de las mismas.

Bibliografía

[1] Posit, "Shiny." https://shiny.posit.co/.

[2] The R Foundation, "R." https://www.r-project.org/.

[3] Software Freedom Conservancy, "Selenium Webdriver." https://www.selenium.dev/documentation/webdriver/.

[4] World Wide Web Consortium., "W3c." https://www.w3.org/.

[5] Ramon Diaz-Uriarte and Pablo Herrera-Nieto, "EvAM-Tools." https://www.iib.uam.es/evamtools/.

[6] Software Freedom Conservancy, "Selenium." https://www.selenium.dev/.

[7] Ramón Diaz Uriarte, "EvAM-Tools: Methods." https://rdiaz02.github.io/EvAM-Tools/pdfs/evamtools_methods_details_faq.pdf.

[8] Ramón Diaz Uriarte, "EvAM-Tools: Examples." https://rdiaz02.github.io/EvAM-Tools/pdfs/evamtools_examples.pdf.

[9] Google, "Puppeteer." https://pptr.dev/.

[10] microsoft, "Playwright." https://playwright.dev/.

[11] Cypress.io, "Cypress." https://www.cypress.io/.

[12] Angular, "Protractor." https://www.protractortest.org/#/.

[13] Appium, "Appium." https://appium.io/docs/en/2.0/.

[14] 2GIS, "Winium." https://github.com/2gis/Winium.

[15] Microsoft, "WinAppDriver." https://github.com/microsoft/WinAppDriver.

[16] Microsoft, "Microsoft UI Automation." https://learn.microsoft.com/en-us/dotnet/framework/ui-automation/ui-automation-overview.

[17] Smartbear, "TestComplete." https://smartbear.com/product/testcomplete/.

[18] Python Software foundation, "Python." https://www.python.org/.

[19] Mozilla Corp., "Firefox ESR." https://support.mozilla.org/es/kb/cambiar-firefox-extended-support-release-esr-para-uso-personal.

[20] Mozilla Corp., "Geckodriver." https://github.com/mozilla/geckodriver/releases.

[21] Mozilla Corp., "Mozilla Corporation." https://www.mozilla.org/es-ES/contribute/.

[22] Awio web services, "W3Counter." https://www.w3counter.com/globalstats.php.

[23] Python Software foundation, "PyPi." https://pypi.org/.

[24] D. Crockford, "JavaScript Object Notaion." https://www.json.org/json-es.html.

[25] Python Software foundation, "unittest." https://docs.python.org/3/library/unittest.html.

[26] Baiju Muthukadan, “ActionsActionChains.” https://selenium-python.readthedocs.io/api.html?highlight=actionchai#selenium.webdriver.common.action_chains.ActionChains.