Automation of Test Evolution

Burak - Karakoç karakoc@sabanciuniv.edu

Computer Science and Engineering, Junior

Muhammed Orhun - Gale (Student) morhun@sabanciuniv.edu

Computer Science and Engineering, Junior

Cemal - Yılmaz

Computer Science and Engineering

Anıl - Koyuncu

Computer Science and Engineering

Abstarct

As the technologies and the expectations on a software application are enormously increasing day by day, the time and labor required to stabilize the end-to-end testing of a product are increasing as well. In other words, automation of testing becomes much more important. The key to efficient, less resource-dependent test automation is to have AI/ML-aided testing systems. In this project, our ambition is to develop a learning-based approach that understands changes in the application interface and updates the test cases accordingly to ensure that the system under test is performing as designed.

Keywords — Artificial Intelligence, the Software Testing, Test Automation.

Introduction

Testing is an important part of successful product development. As the technologies and the data become more and more complex, the automation of software testing has become a must either. Further, to automate the software testing efficiently, AI and ML techniques should be implemented and integrated into the testing system properly. According to H. Hourani, A. Hammad, and M. Lafi (2019), as the software applications are being more complex, time is becoming a critical factor to release applications that must be end-to-end tested and comply with the business requirements. And artificial intelligence is a key to achieving more accurate results therefore saving time. (pp. 565-569)

End-to-end (E2E) testing is a technique that tests the entire software product from beginning to end to ensure the application flow behaves as expected. The main objective is to test from the end user's experience by simulating the real user scenario and validating the system under test and its components. Unfortunately, end-to-end tests are particularly fragile as any change in the application interface, e.g., application flow, location, or name of graphical user interface (GUI) elements, requires a change in the tests.

In this sense, designing E2E testing systems which are able to adapt themselves to changes and anomalies in on-test softwares or creating new tests/test approaches by learning from previous tested softwares can be seen as one of the important and emerging aspect of Software Testing. Creating such evolving systems might be possible with help of the Artificial Intelligence. (Chen et al., pp. 1-3) By developing approaches which employ machine learning algorithms such as Q-Learning, it is possible to create testing systems that tests different properties of a software such as robustness. (Bauersfeld and Vos, 2012, pp. 1-4) There are a variety of possible strategies that could be apply to create such systems. For instance, automatized analyze of code scripts to detect and interpret changes in the software via using NLP could be a fruitful approach. Another approach can be detection of the changes in the GUI via using image processing and computer vision techniques. Detection of GUI elements, texts and related new GUI pages/sites could be extracted via using computer vision and deep learning models. (Chen et al., 2020, pp. 2-6)

In this project, our aim is to develop an E2E software testing system that can analyze GUI of an on-test software and generate an automatized test scene for it.

End-to-End Software Testing with Computer Vision

Fundamentally, the E2E software testing system that is developed in this project uses various computer vision algorithms to detect texts, so all clickable and unclickable objects, on the GUI of the web site and returns their locations to user to create a test case via using these locations. In this process, first, from UI of the program, which algorithm would be used is selected by tester and after that a browser is launched. In the browser, tester is allowed to surf freely until he selects the analyze option. After analyze option is pressed by the tester, screenshot of the web site is taken fully or partially, and web site is analyzed by the previously selected computer vision algorithm. After analyze process ends, found key words and their locations are displayed in the UI and lets user to search a specific word. After the tester decides to which objects (words) will be clicked in the test and creates the test scenario accordingly, automatized test begins in a newly launched browser.

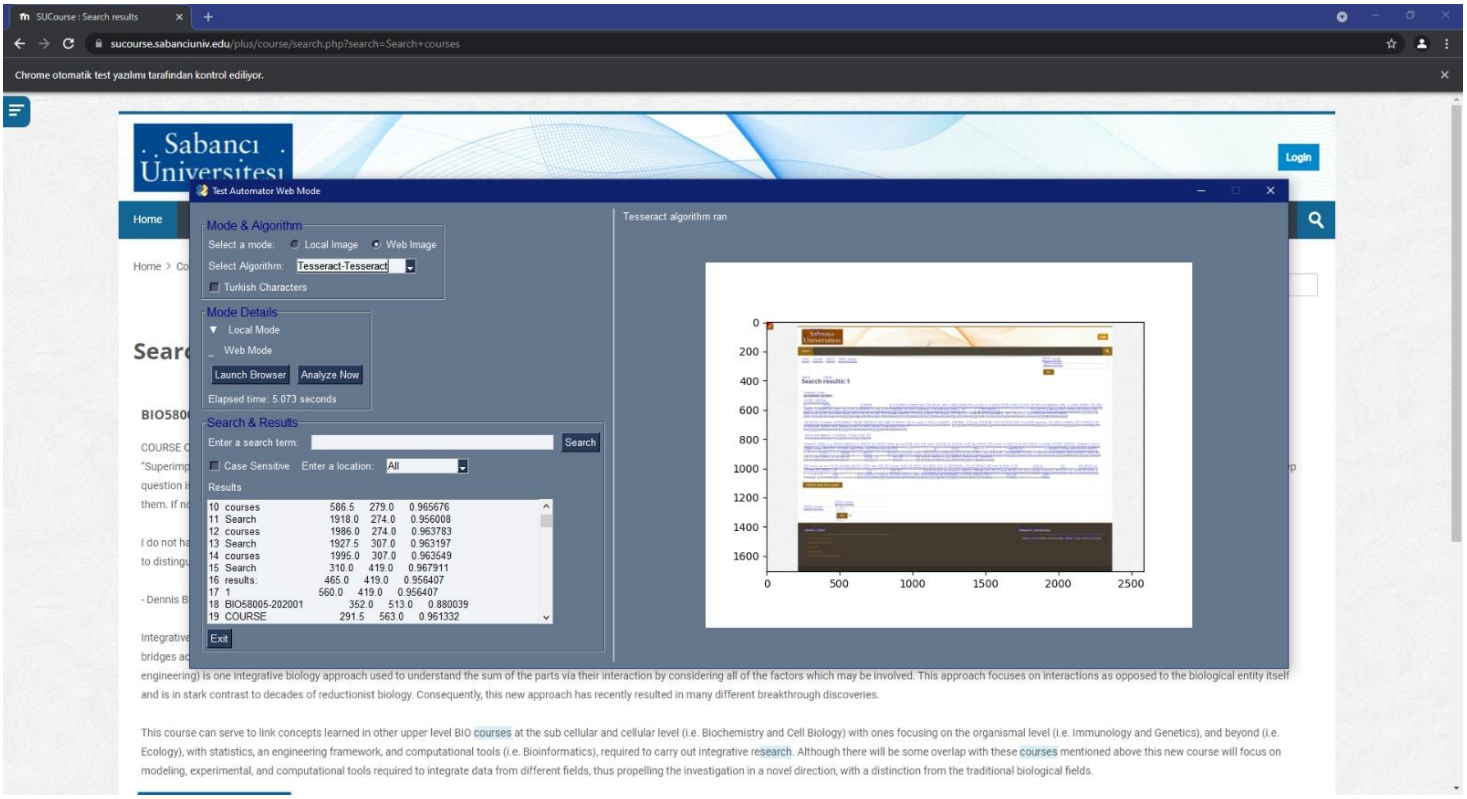


Figure 1 – UI of the System

Computer Vision algorithms that we have used are:

1. Tesseract
2. easyOCR
3. ResNet

Our program is separated into two parts, which are character detection and character recognition. The left side of the algorithm names indicates the text detection done by using that engine and the right side means text recognition is done by using that engine. We derived those algorithms to see whether some might be better at detecting but not recognizing better than others or opposite, especially if the language is not English.

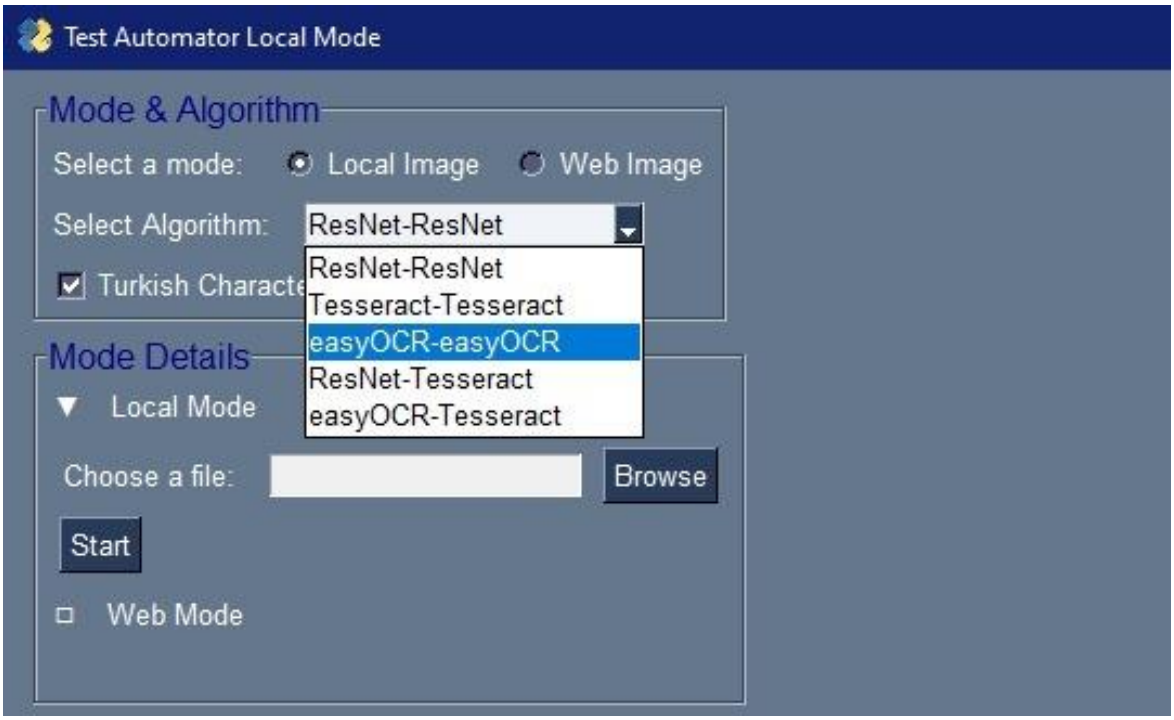


Figure 2 – Algorithm Selection

The principles of text detection and recognition processes are almost the same at three of the algorithms, however, the engine that they are using, therefore built-in methods are completely different than each other. The algorithms are based on finding the bounding boxes with the built-in methods of that OCR engine that we are using on that algorithm, such as image_to_data, or image_to_boxes methods of Tesseract Engine. We find the proper ways to draw the bounding boxes of every clickable and non-clickable object on the screen and retrieve some metrics from those boxes.

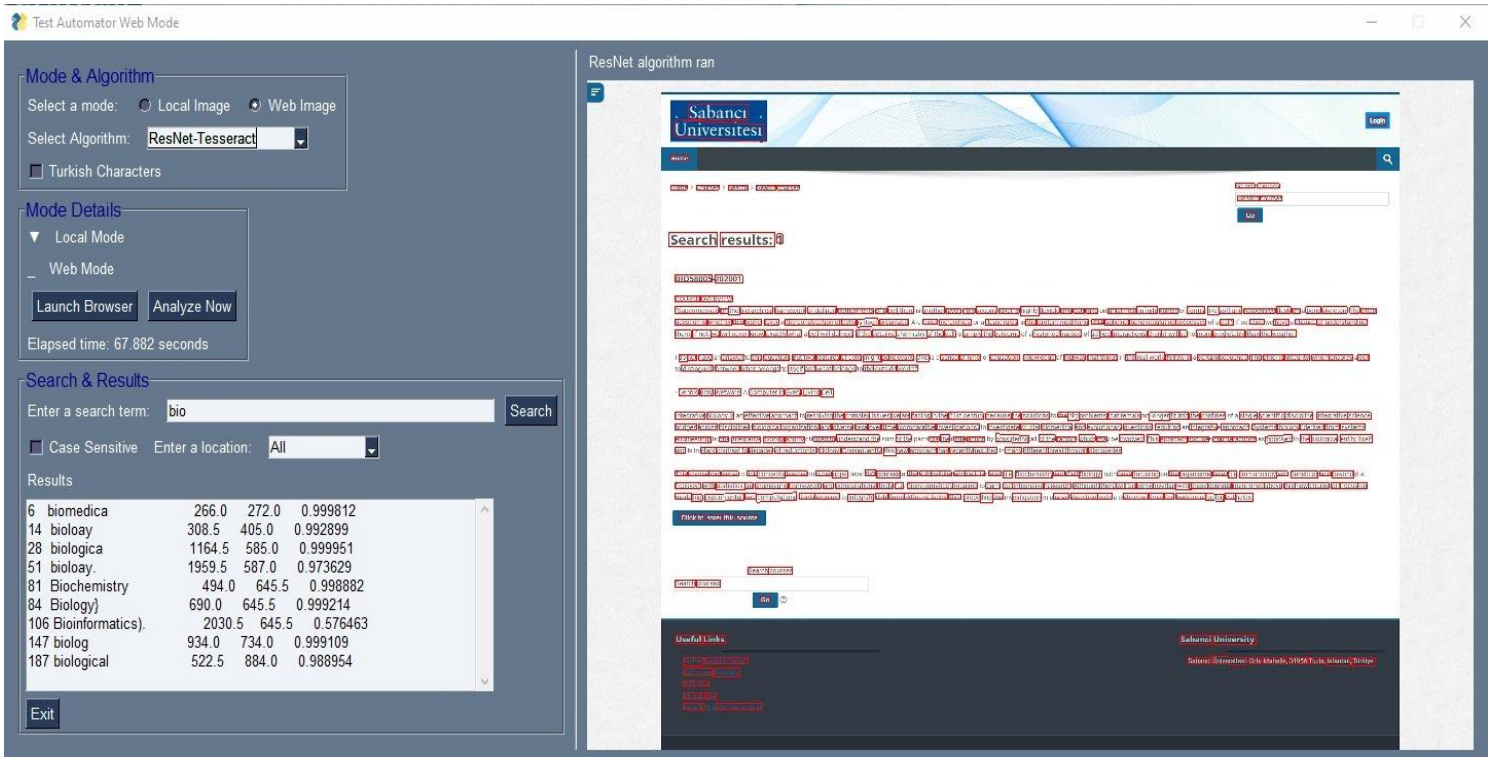


Figure 3.a – ResNet-Teserract

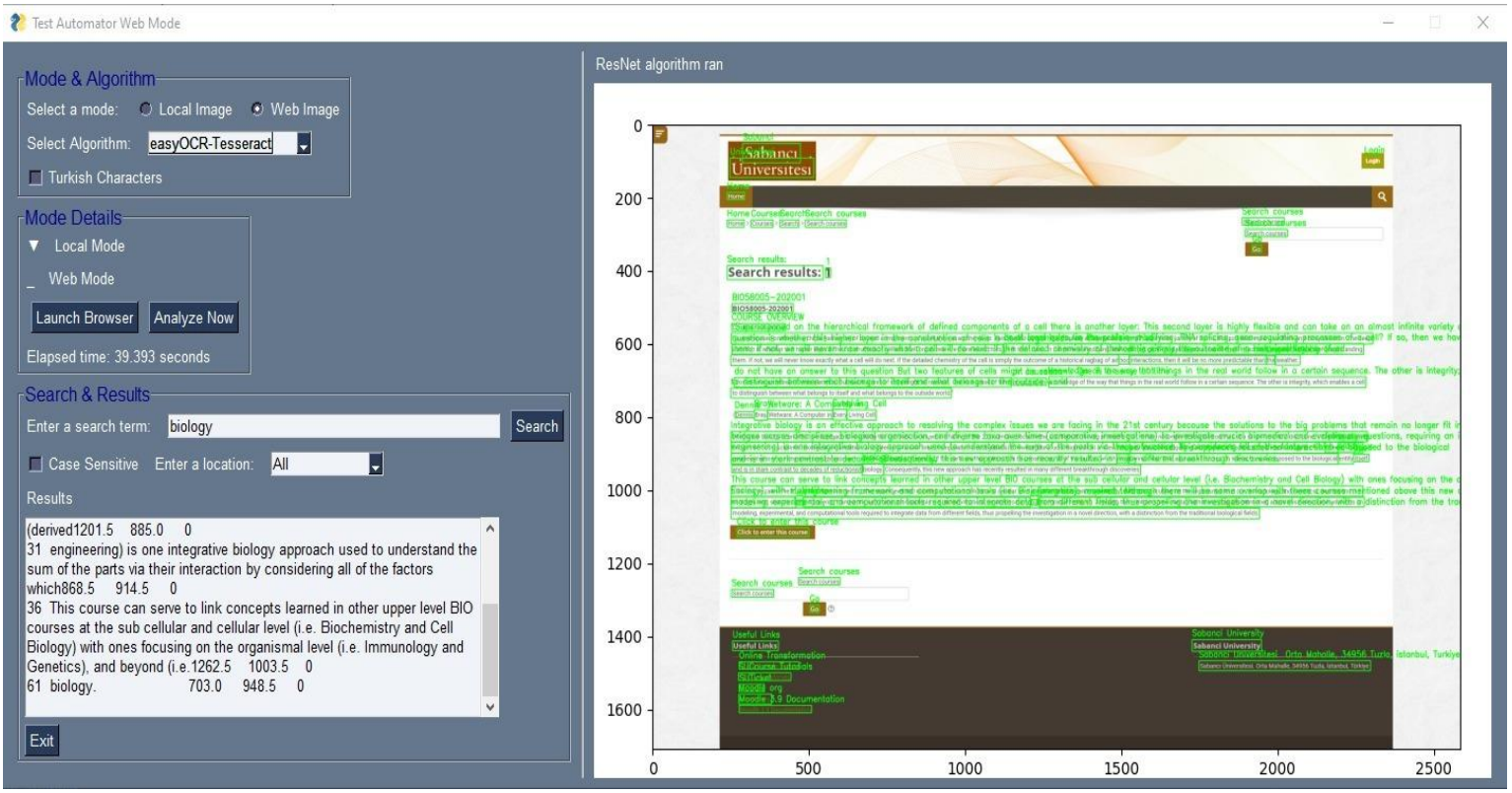


Figure 3.b – easyOCR-Teserract

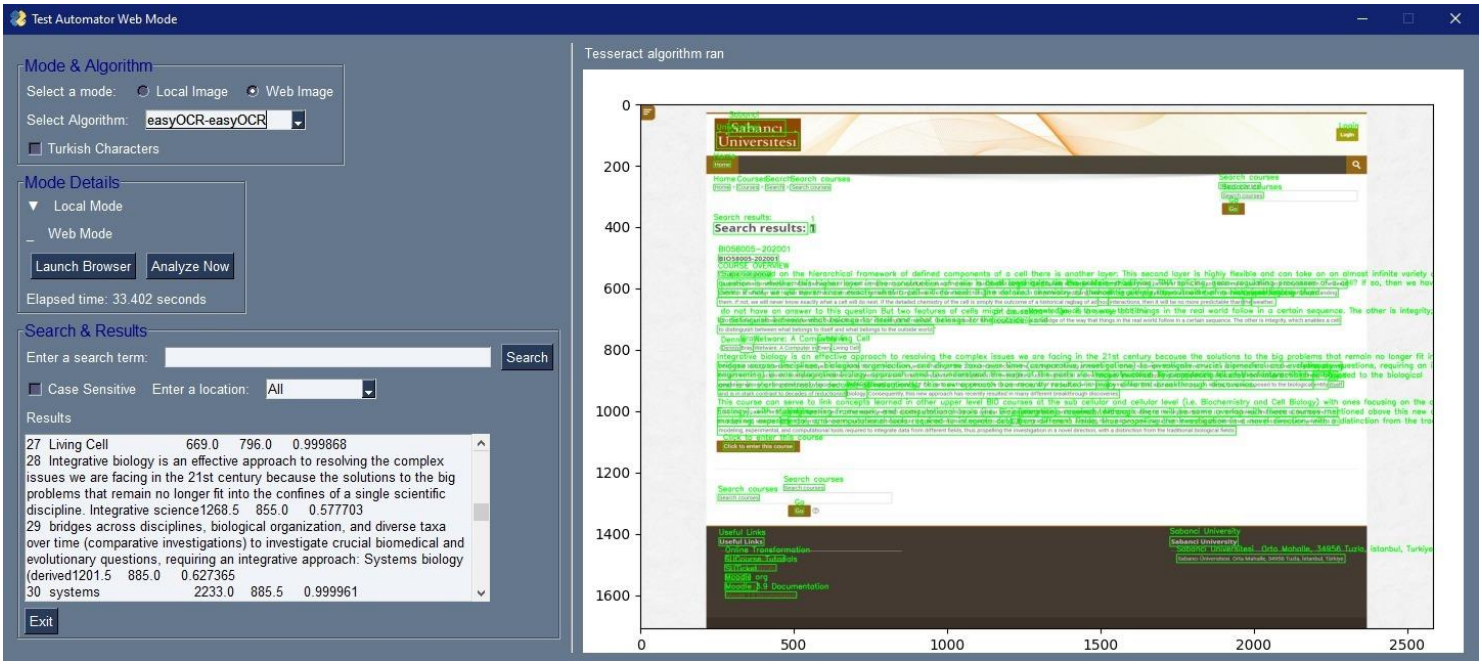


Figure 3.c – easyOCR-easyOCR

The metrics that we achieve and store are: order, predicted text, confidence score, dimensions (width-height), location (top-right, bottom-left), and the center point of the image (to click the item if needed). After retrieving those objects (texts) with their properties, we basically store them as JSON format in a map to achieve them in an efficient way (when the user i.e. test, wants to find the object).

We decided to test our algorithms as well. In order to do that, we decided to develop a module that stores the previous search keys, parses the DOM (from the searched screen), and finds its original locations to compare their values. Parsing operation is done by using python and selenium. After parsing the matching items, their locations were retrieved and stored in JSON format. By doing those operations, we were able to compare the actual key and the key that returned from the current algorithm that we are using.

Previously analyzed web window (Selenium ChromeDriver tab):

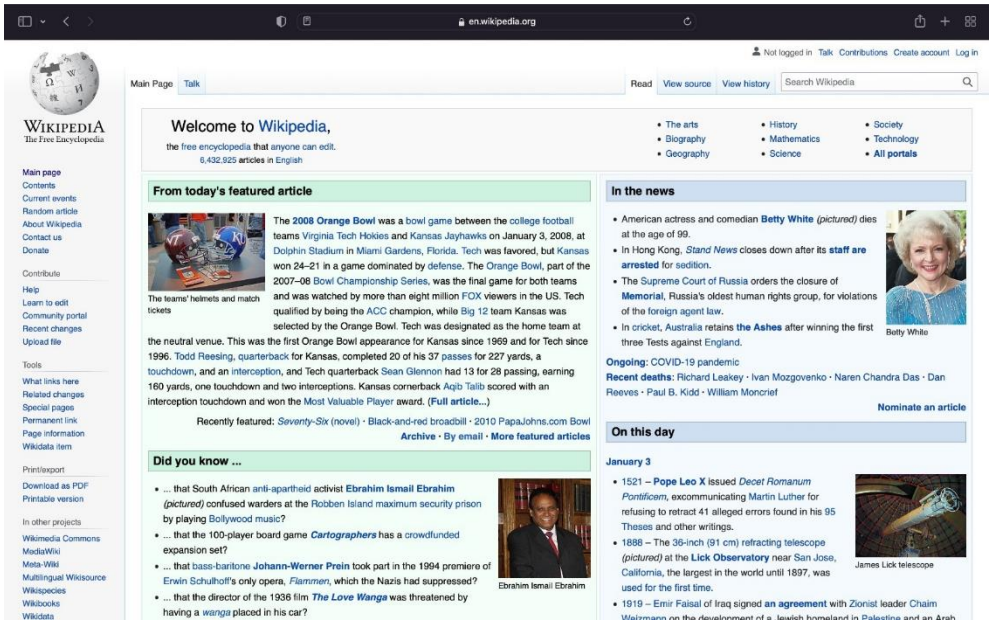


Figure 4.

‘The arts’, ‘Welcome to’, ‘test sentence not to found’ are searched previously, and the results from the DOM parsing below:

```
Cannot find [test sentence not to found]
{'name': 'The arts', 'x': 819, 'y': 118, 'height': 15, 'width': 49}
{'name': 'Welcome to', 'x': 208, 'y': 107, 'height': 41, 'width': 308}
```

Figure 5.

For the automatized testing part of the project, Cucumber which is a software tool that supports Behavior-driven development is used to create E2E testing scenarios. (Fig. x) Actions that can be taken are as follows ‘launch chrome browser’, ‘Analyze screen: "Full/Partial”’, ‘click "Keyword" on the screen’, ‘Reset Screen’ and ‘close browser’

```
Feature: Clickable
  Scenario: Clicking to clickable places
    Given launch chrome browser

    When Analyze screen: "Full"
    When click "Enerji" on the screen
    When Reset Screen

    When Analyze screen: "Partial"
    When click "Sabanci" on the screen
    When click "Muhendislik" on the screen
    When Reset Screen

    Then close browser
```

Figure 6– An example Cucumber BDD scenerio

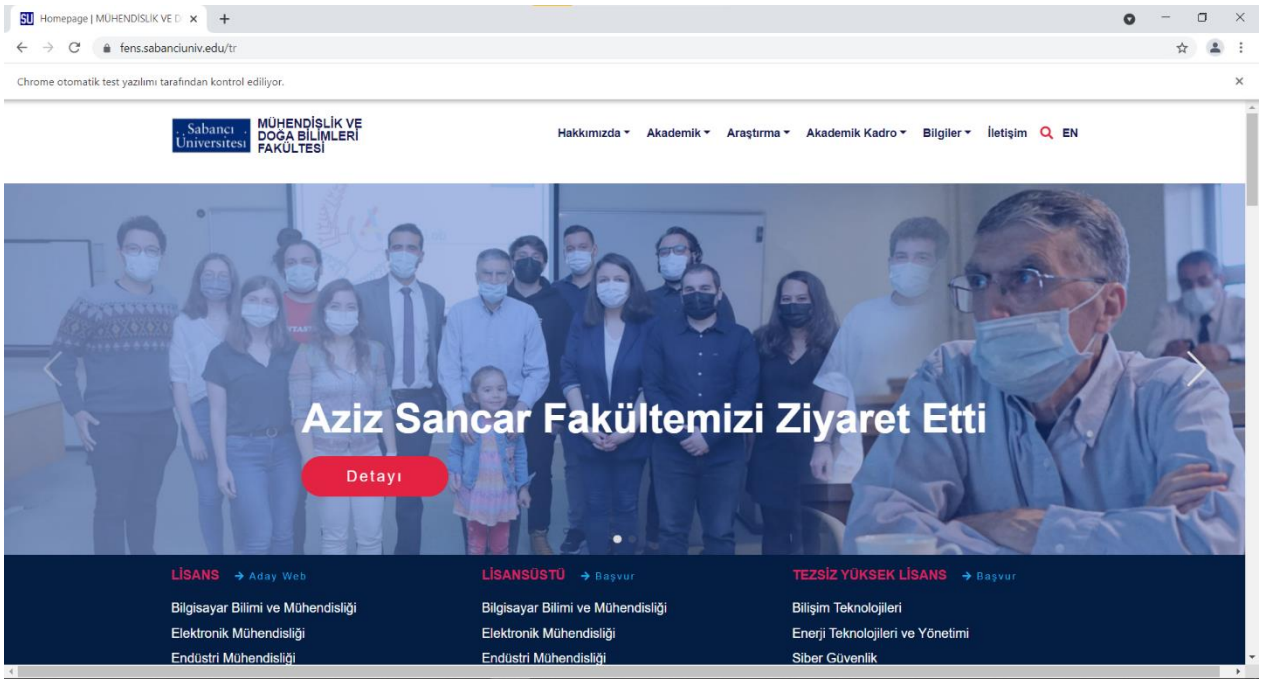


Figure 7 – An on-test web site

Functions that are used in these scenarios are created by using Selenium, which is a web test automatization tool. Selenium is used to perform opening and closing browsers, taking screenshots, and clicking to specified locations. Even though any of the desired functionality is met by the tools that Selenium provided, for some implementations those tools fell behind the needs.

```
def chrome_takeFullScreenshot(driver):
    def send(cmd, params):
        resource = "/session/%s/chromium/send_command_and_get_result" % driver.session_id
        url = driver.command_executor._url + resource
        body = json.dumps({'cmd': cmd, 'params': params})
        response = driver.command_executor._request('POST', url, body)
        return response.get('value')

    def evaluate(script):
        response = send('Runtime.evaluate', {'returnByValue': True, 'expression': script})
        return response['result']['value']

    metrics = evaluate(
        "({" +
        "width: Math.max(window.innerWidth, document.body.scrollWidth, document.documentElement.scrollWidth)|0," +
        "height: Math.max(innerHeight, document.body.scrollHeight, document.documentElement.scrollHeight)|0," +
        "deviceScaleFactor: window.devicePixelRatio || 1," +
        "mobile: typeof window.orientation !== 'undefined'" +
        "})")
    send('Emulation.setDeviceMetricsOverride', metrics)
    screenshot = send('Page.captureScreenshot', {'format': 'png', 'fromSurface': True})
    send('Emulation.clearDeviceMetricsOverride', {})

    return base64.b64decode(screenshot['data'])
```

Figure 8 – Full page screenshot function

To implement screenshot-taking function, first, functions as ‘getScreenshot()’ are used however, implementations that Selenium provides are not satisfying for the full page screenshots because of the low resolution. Therefore, an adequate screenshot function is implemented (Fig. x+2). Another problem that was encountered is the clicking to the exact locations that are provided by the computer vision algorithms on the browser. Problem is stemmed from the fact that every tester can use their computer with different resolutions and also, they may change the size of the tested screen while test process is going on. This problem is handled via scaling the locations that are provided by the computer vision algorithms according to the sizes of the screenshot which was used and size of the screen which was currently in use.

Discussion and Conclusion

In this project, it is observed that conducting E2E testing with the help of Artificial Intelligence makes testing attempts not only easier but also it makes the testing process more efficient in terms of the exploration of possible missing points since overlooking is not likely for a computer vision algorithm when it is assumed that algorithm works correctly. Here, GUI is analyzed in a way to find the texts and their locations on the screen. This approach can be strengthened via elaborating

the information in the code script or employing some learning algorithm which provides the information that which object is clickable by analyzing only the GUI appearance.

References

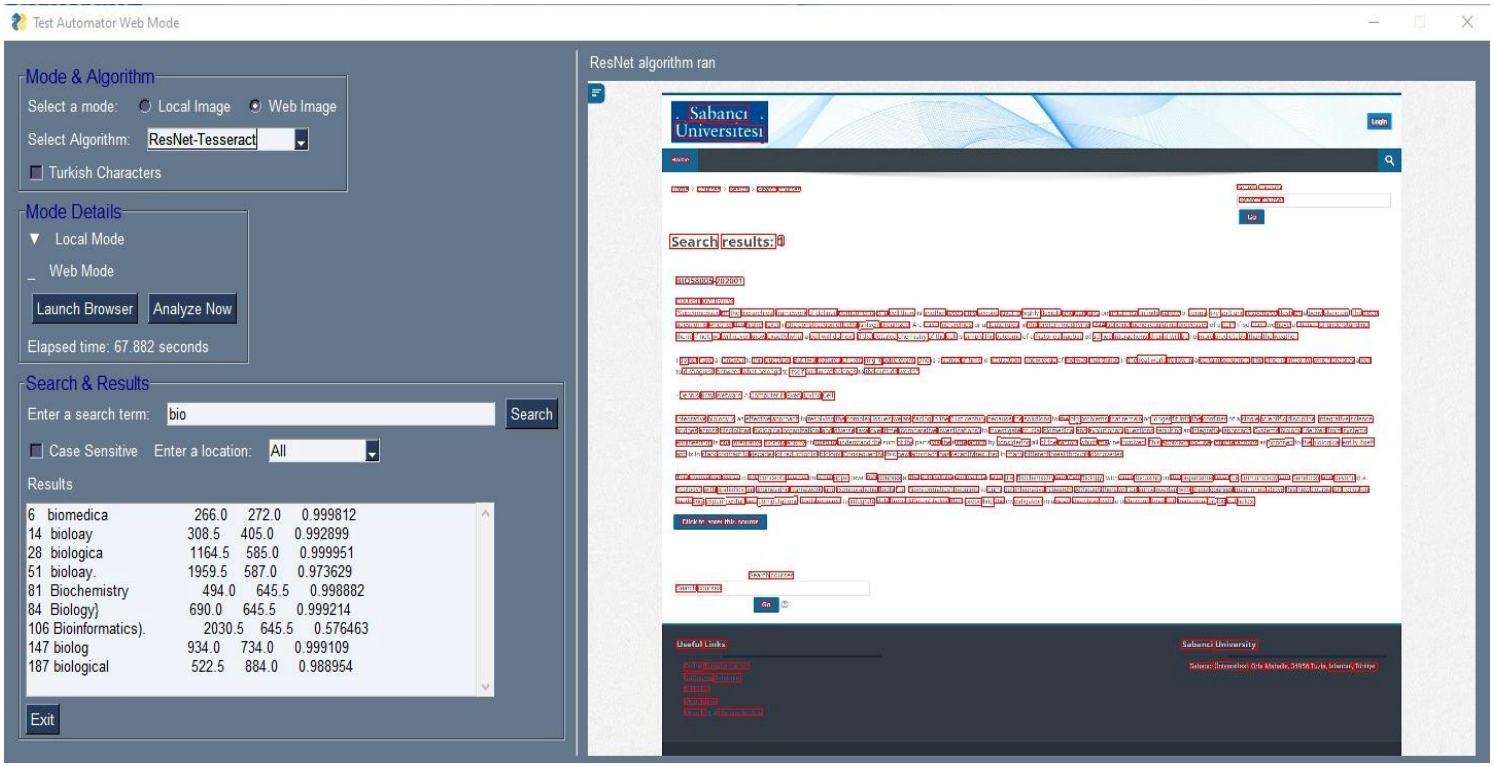
[1] H. Hourani, A. Hammad and M. Lafi, "The Impact of Artificial Intelligence on Software Testing," 2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), 2019, pp. 565-570, doi: 10.1109/JEEIT.2019.8717439.

[2] Chen, J., Xie, M., Xing, Z., Chen, C., Xu, X., Zhu, L., & Li, G. (2020). Object detection for graphical user interface: old fashioned or deep learning or a combination? Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

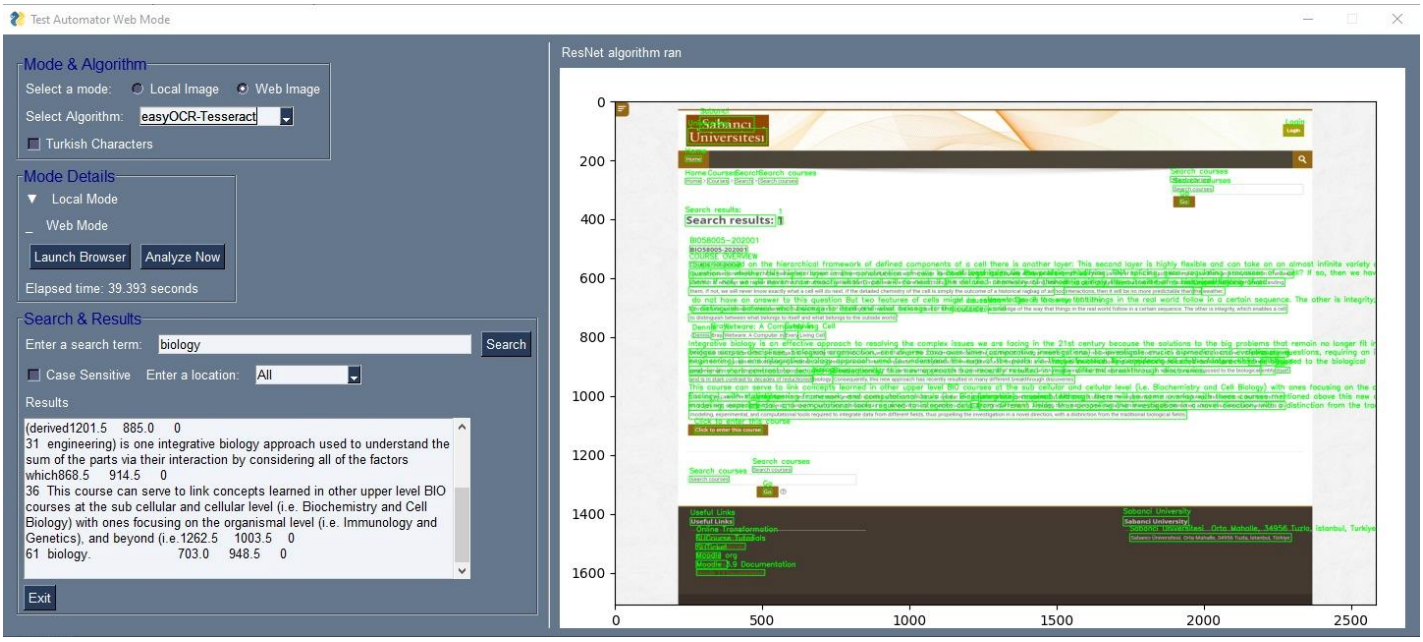
[3] Bauersfeld, S., & Vos, T.E. (2012). A Reinforcement Learning Approach to Automated GUI Robustness Testing.

Appendices

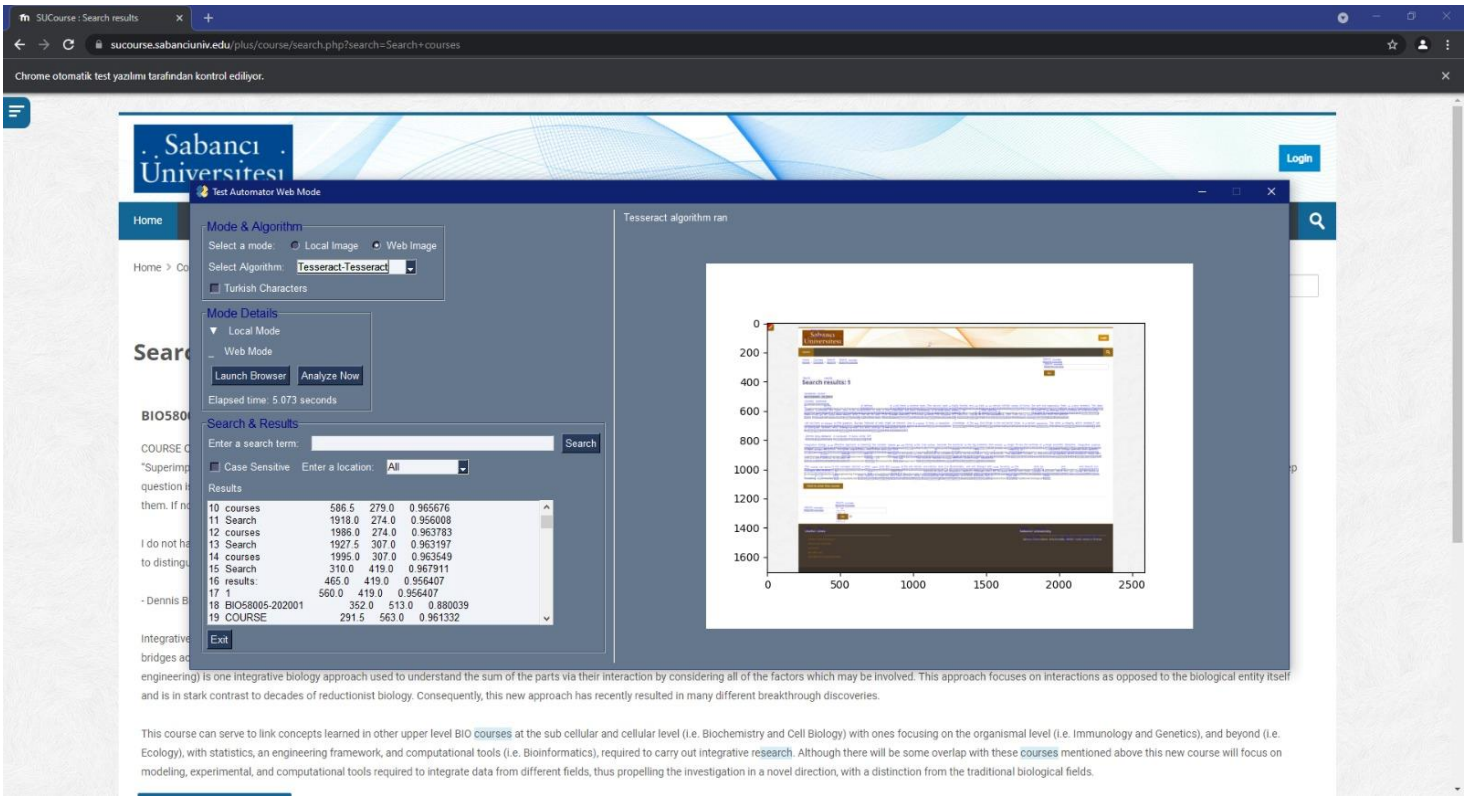
Appendix 1



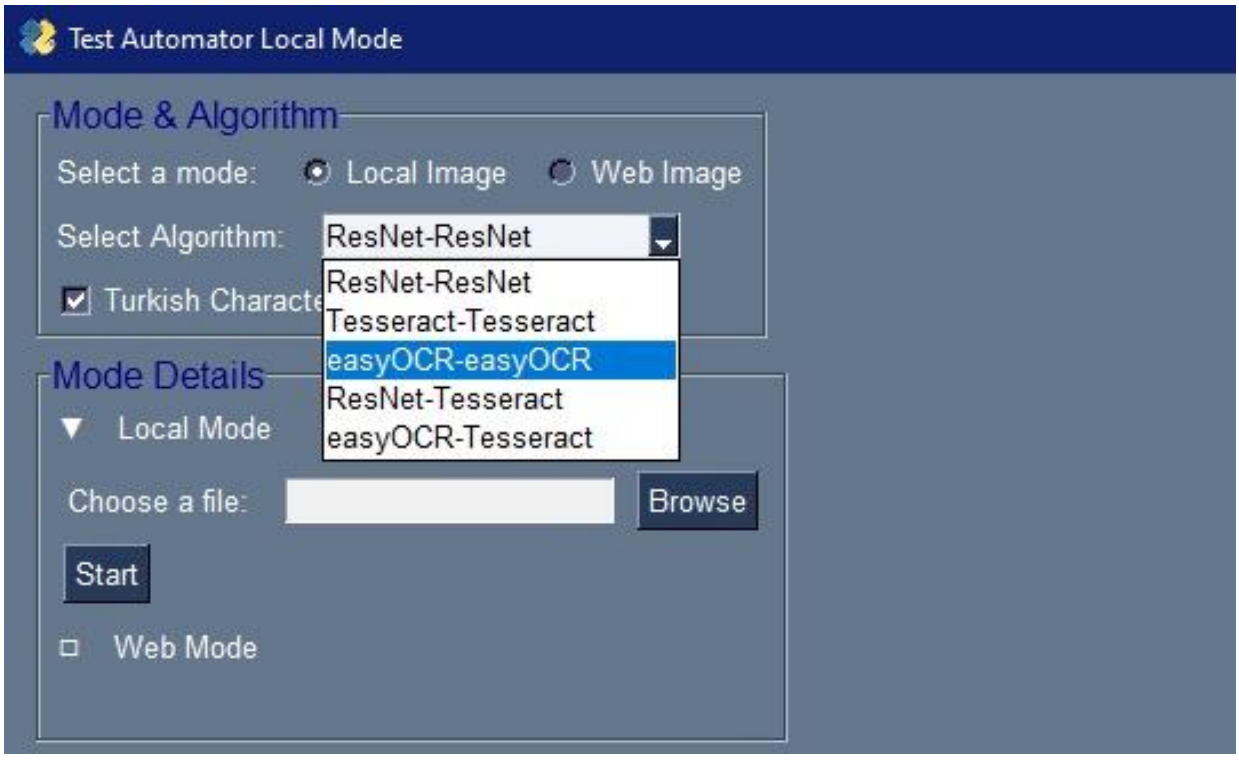
Appendix 2



Appendix 3



Appendix 4



Appendix 5

```
Feature: Clickable
  Scenario: Clicking to clickable places
    Given launch chrome browser

    When Analyze screen: "Full"
    When click "Enerji" on the screen
    When Reset Screen

    When Analyze screen: "Partial"
    When click "Sabanci" on the screen
    When click "Muhendislik" on the screen
    When Reset Screen

    Then close browser
```

Appendix 6

```
# Define click action
ac = ActionChains(context.driver)

# Click given locations
print("Width x Height:", width, "x", height)
i = 0
for rows in matrix:
    ac.move_to_element_with_offset(context.driver.find_element_by_tag_name('body'), 0, 0)
    # Positions that are detected by the CV algorithms are related with the screenshot's resolution
    # For the web analyze, it is okay since we control the resolution
    # However, for the local, we should adapt location according to the given ss's resolution
    print(matrix[i][0], matrix[i][1])
    ac.move_by_offset(wScale*matrix[i][0]+browser_location["x"], hScale*matrix[i][1]+browser_location["y"]).click().perform()
    sleep(1)
    i = i + 1
```

Appendix 7

```
def chrome_takeFullScreenshot(driver):
    def send(cmd, params):
        resource = "/session/%s/chromium/send_command_and_get_result" % driver.session_id
        url = driver.command_executor._url + resource
        body = json.dumps({'cmd': cmd, 'params': params})
        response = driver.command_executor._request('POST', url, body)
        return response.get('value')

    def evaluate(script):
        response = send('Runtime.evaluate', {'returnByValue': True, 'expression': script})
        return response['result']['value']

    metrics = evaluate(
        "({" +
        "width: Math.max(window.innerWidth, document.body.scrollWidth, document.documentElement.scrollWidth)|0," +
        "height: Math.max(innerHeight, document.body.scrollHeight, document.documentElement.scrollHeight)|0," +
        "deviceScaleFactor: window.devicePixelRatio || 1," +
        "mobile: typeof window.orientation !== 'undefined'" +
        "})")
    send('Emulation.setDeviceMetricsOverride', metrics)
    screenshot = send('Page.captureScreenshot', {'format': 'png', 'fromSurface': True})
    send('Emulation.clearDeviceMetricsOverride', {})

    return base64.b64decode(screenshot['data'])
```

Appendix 8

```
def chrome_takePartialScreenshot(driver):
    def send(cmd, params):
        resource = "/session/%s/chromium/send_command_and_get_result" % driver.session_id
        url = driver.command_executor._url + resource
        body = json.dumps({'cmd': cmd, 'params': params})
        response = driver.command_executor._request('POST', url, body)
        return response.get('value')

    def evaluate(script):
        response = send('Runtime.evaluate', {'returnByValue': True, 'expression': script})
        return response['result']['value']

    metrics = evaluate(
        "({" +
        "width: 900," +
        "height: 700," +
        "deviceScaleFactor: window.devicePixelRatio || 1," +
        "mobile: typeof window.orientation !== 'undefined'" +
        "})")
    send('Emulation.setDeviceMetricsOverride', metrics)
    screenshot = send('Page.captureScreenshot', {'format': 'png', 'fromSurface': True})
    send('Emulation.clearDeviceMetricsOverride', {})
```

Appendix 9

```
@given('launch chrome browser')
def launchBrowser(context):
    # Open Chrome
    options = webdriver.ChromeOptions()
    options.add_experimental_option('excludeSwitches', ['enable-logging'])
    context.driver = webdriver.Chrome(options=options, desired_capabilities=capabilities)

    global image

    url = 'https://fens.sabanciuniv.edu/tr'

    context.driver.set_window_position(0, 0)
    context.driver.maximize_window()
    context.driver.get(url)
```

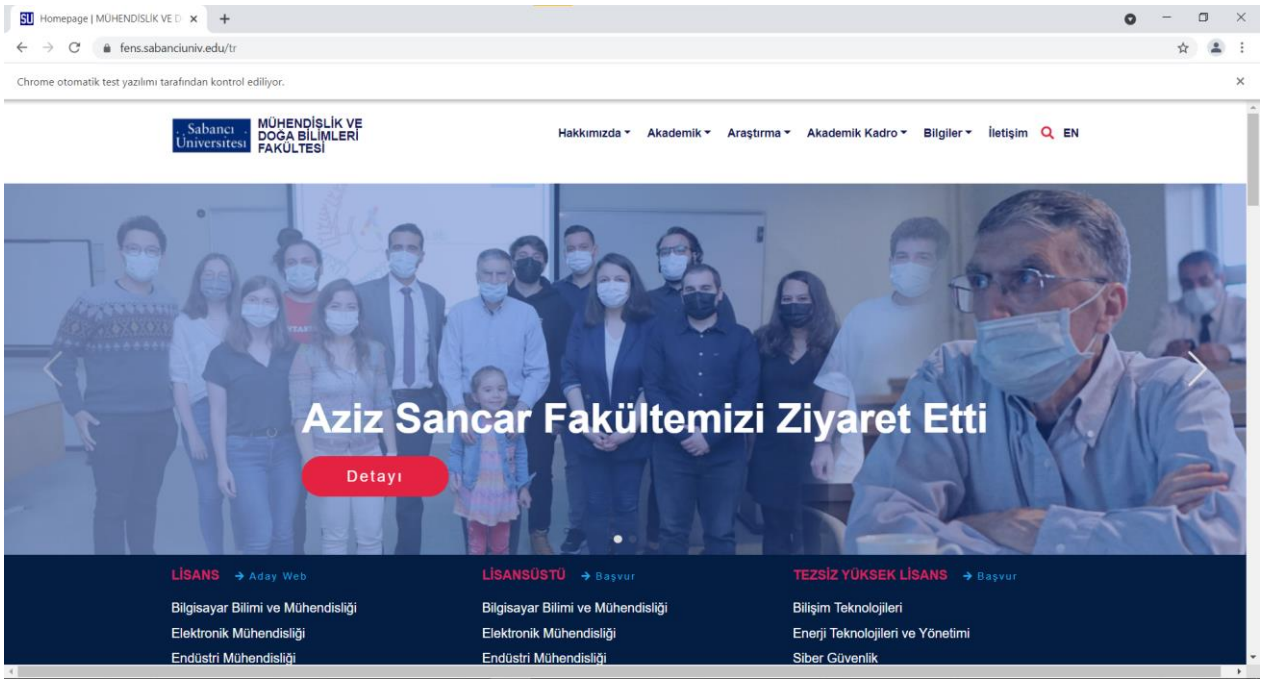
Appendix 10

```
@when('Analyze screen: "{Full_Partial}"')
def testProcess(context, Full_Partial):
    path = "C:/Users/hp/PycharmProjects/pure_test_automation_test2/features/output/sc.png"

    if Full_Partial == "Full" or Full_Partial == "full":
        image = chrome_takeFullScreenshot(context.driver)
        with open(path, 'wb') as f:
            f.write(image)

    else:
        image = chrome_takePartialScreenshot(context.driver)
        with open(path, 'wb') as f:
            f.write(image)
```

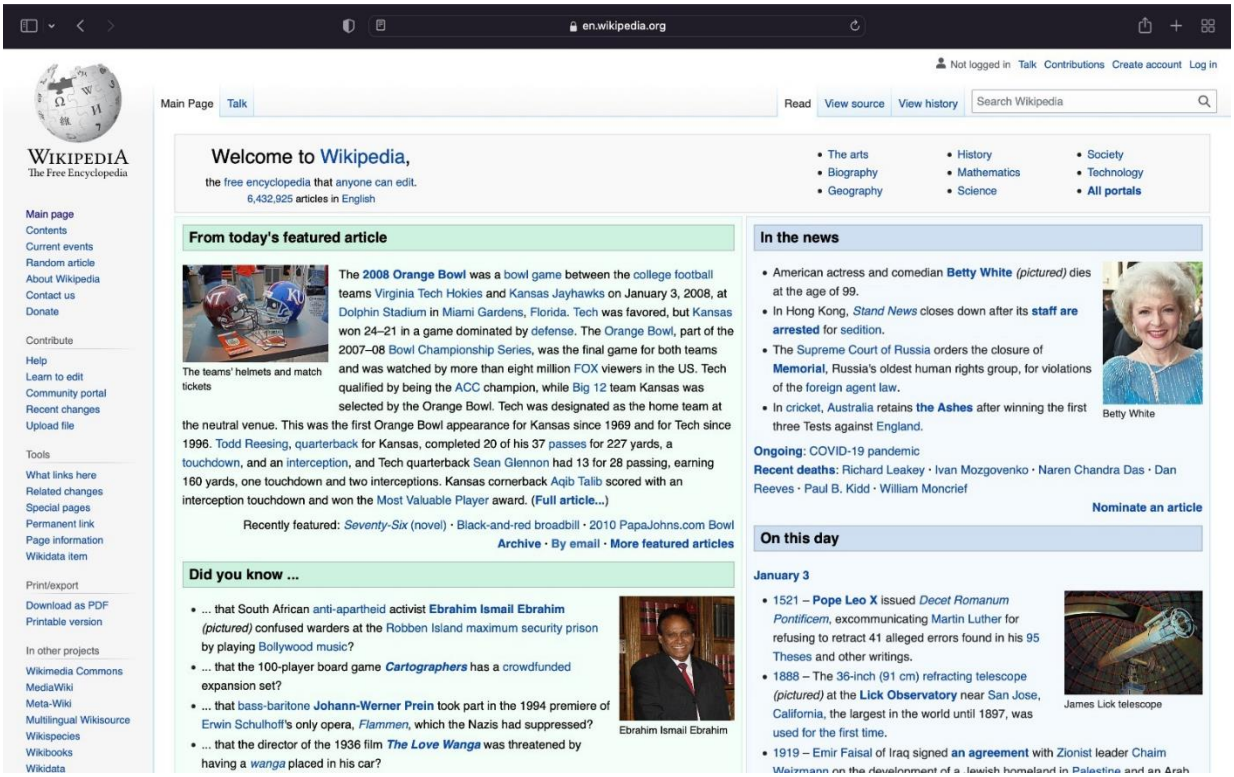
Appendix 11

```
@when('Reset Screen')
def testProcess(context):
    image = 0


@then('close browser')
def closeBrowser(context):
    # Close browser
    context.driver.close()
#    os.remove("C:/Users/hp/PycharmProjects/pure_test_automation_test2/features/output/sc.png")
```

Appendix 12



Appendix 13



Appendix 14

```
Cannot find [test sentence not to found]
{'name': 'The arts', 'x': 819, 'y': 118, 'height': 15, 'width': 49}
{'name': 'Welcome to', 'x': 208, 'y': 107, 'height': 41, 'width': 308}
```