Final Report – Automated Script Generation (Katalon)

Project Repository (link)

https://github.ubc.ca/mds-cl-2021-22/katalon_project/tree/master

Outline

1. Description

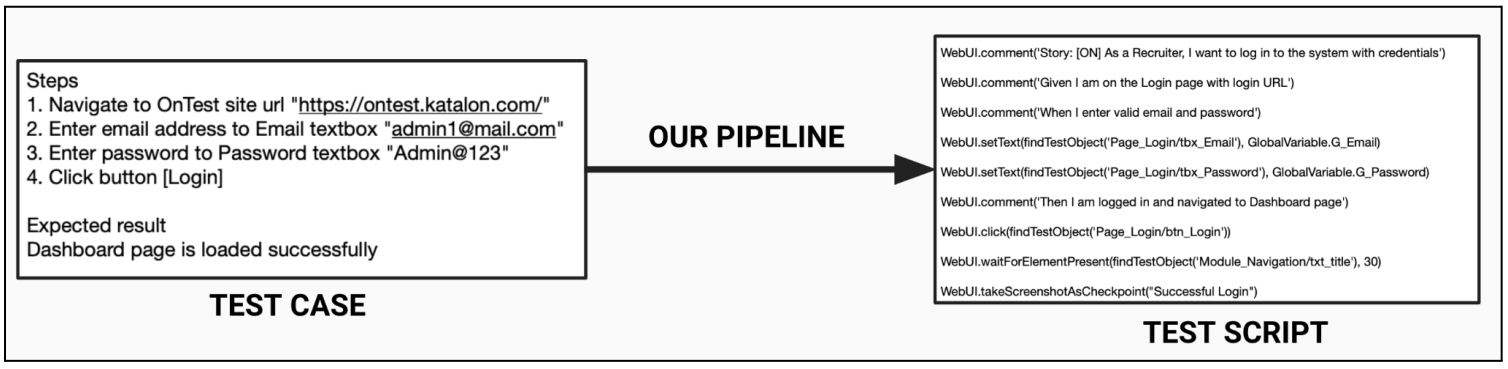
Before deploying software products in the commercial space, developers need to ensure their products perform as intended. The process of verifying or evaluating their software products is known as software testing. A rather candid way of testing software would be to simply use it to identify any faults in the product. A more sophisticated and elaborate version of this is referred to as manual testing, where the tester follows a bunch of steps to verify the performance of the software for different scenarios/use-cases. An alternate and more efficient way of achieving the same purpose is automation testing, effectuated through automation software that uses programmable scripts. Katalon Studio, developed by Katalon Inc., is one such software tool.

Here's an example of a simplified test script, written in the Groovy language, that automates the process of logging into the Katalon software and verifies whether the log-in is successful or not.

```
WebUI.comment('Story: [ON] As a Recruiter, I want to log in to the system with credentials')

WebUI.comment('Given I am on the Login page with login URL')

WebUI.comment('When I enter valid email and password')

WebUI.setText(findTestObject('Page_Login/tbx_Email'), GlobalVariable.G_Email)

WebUI.setText(findTestObject('Page_Login/tbx_Password'), GlobalVariable.G_Password)

WebUI.comment('Then I am logged in and navigated to Dashboard page')

WebUI.click(findTestObject('Page_Login/btn_Login'))

WebUI.waitForElementPresent(findTestObject('Module_Navigation/txt_title'), 30)

WebUI.takeScreenshotAsCheckpoint("Successful Login")
```

For this project, our objective was to help Katalon Studio in building an end-to-end pipeline that can automatically generate scripts that facilitate automation testing, from a list of steps for each possible scenario to test, collectively referred to as test cases, provided in natural language (English in this case).

Simply put, we aim to build a pipeline that can automatically translate manual test use-cases for a software, written in English, into an actionable script, which accelerates the process of automation testing and saves a lot of valuable time for the testers. Below is an illustration that encapsulates the gist of the pipeline we intend to build for the project.



2. Datasets

The two main components of the provided dataset are manual test cases and test scripts. Here is an example.

Manual test case:

```
Steps
1. Navigate to OnTest site url.
2. Enter email address to Email textbox admin1@mail.com
3. Enter password to Password textbox “Admin@123”,
4. click button [Login].
Expected result
The dashboard page is loaded successfully.
```

(Simplified) Test script:

```
WebUI.setText(findTestObject('Page_Login/tbx_Email'), GlobalVariable.G_Email)
WebUI.setText(findTestObject('Page_Login/tbx_Password'),
GlobalVariable.G_Password)
WebUI.click(findTestObject('Page_Login/btn_Login'))
WebUI.waitForElementPresent(findTestObject('Module_Navigation/txt_title'),
30)
WebUI.takeScreenshotAsCheckpoint("Successful Login")
```

Full manual test cases are stored in a CSV file. Each manual test case comes with these attributes: *Feature,* the functionality that needs to be tested, *ID no.,* the test case id number, *Full manual test cases,* listed test instructions. The test cases mainly contain three sections, *Pre-condition,* setups for the test, *Steps,* test instructions, and *Expected Results,* expected output from the test.

A	B	C	D	E	F	G
Feature	ID.N	Title	Test Scenario	Full manual test cases	TC Priori	Automat
Create Libraries	TC010	TC010_Verify Unsuccessful Create Library In Empty Page_P4	Verify login successfully with valid credentials	Pre-condition - All the libraries in system have been deleted - User has logged in to OnTest successfully (refer TC-001) Steps 1. From the navigation bar on the left site, click Libraries and should see notice "You have no librabries yet" 2. Click button [Create Library] 3. On the dialog Create Librabry, leave Librabry Name empty 4. Click button [Submit] Expected result - Error message displays 'This field is required.' for Librabry Name	P4	Yes

Katalon provided 128 test scripts in total which are stored as files in corresponding Katalon private Github folders. The test scripts are written in Groovy, a programming language that comes with great support for writing tests. We can access the test scripts through Katalon's private Github repo or Katalon studio.

Since the test cases and test scripts are stored separately, we need to integrate them during data extraction. The test scripts were first extracted from the Katalon

repository directly, then each test script was paired up with its corresponding manual test case that was stored in an excel sheet. We added a new column in the excel sheet to store the aligned test scripts and the final version of the extracted data is stored in CSV format.

3. Deliverables

The final deliverable for our project is that we should have an end-to-end pipeline that could generate the code scripts based on the full manual test cases which are composed of pre-condition, steps, or expected results.
For instance, we could face a test scenario to verify if the login is successful with valid credentials. For instance, below is an example of the test case steps:

```
Steps
1. Navigate to OnTest site url.
2. Enter email address to Email textbox admin1@mail.com
3. Enter password to Password textbox “Admin@123”,
4. click button [Login].
Expected result
The dashboard page is loaded successfully.
```

Then our pipeline would generate the corresponding Groovy code for the English instructions.
Our final deliverable is an end-to-end pipeline that can automatically generate scripts that facilitate automation testing, from a list of steps for each possible scenario to test, collectively referred to as test cases, provided in natural language . To process the test case instructions and handle different variations that exist, generalizable NLP modules can reliably extract the action from each instruction, and a sequence labeling model trained on relevant data ensures that our pipeline extracts information about each action correctly when presented with all possible manual test case inputs. This combination provided us with the robustness in the face of noisy data while keeping task-specific linguistic knowledge in mind.

4. Methods

Data Preprocessing

The first thing we do in our project is to extract all relevant information from the data sources provided and gather it in the same place. For the test scripts, we compiled a crawler that takes all the test scripts from Katalon’s script repository. Then, we integrate those scripts into the manual test case excel sheet, aligning each test script with its corresponding test case according to their IDs.

From the excel sheet, we then convert our data into programmable formats. For manual test cases, this means the form of a list of lists of strings. Each of the inner lists corresponds to one of the test cases, and each string within that list corresponds to an instruction from that test case. After collecting data from all test cases, we obtain a list of lists.

```
[['Enter email address to Email textbox "admin1@mail.com"',
  'Enter password to Password textbox "Admin@123"',
  'Click button [Login]',
  'Wait title to be present for (30) seconds'],
 ['Enter email address to Email textbox "invalid@wrong"',
  'Enter password to Password textbox "invalidpassword"',
  'Click [Login] button',
  'Wait email error message to appear for (3) seconds',
  'Get email error message',
  'Check if actual error message is correct',
  'Wait password error message to appear for (3) seconds',
  'Get password error message',
  'Check if actual password message is correct'],
 ...]
```

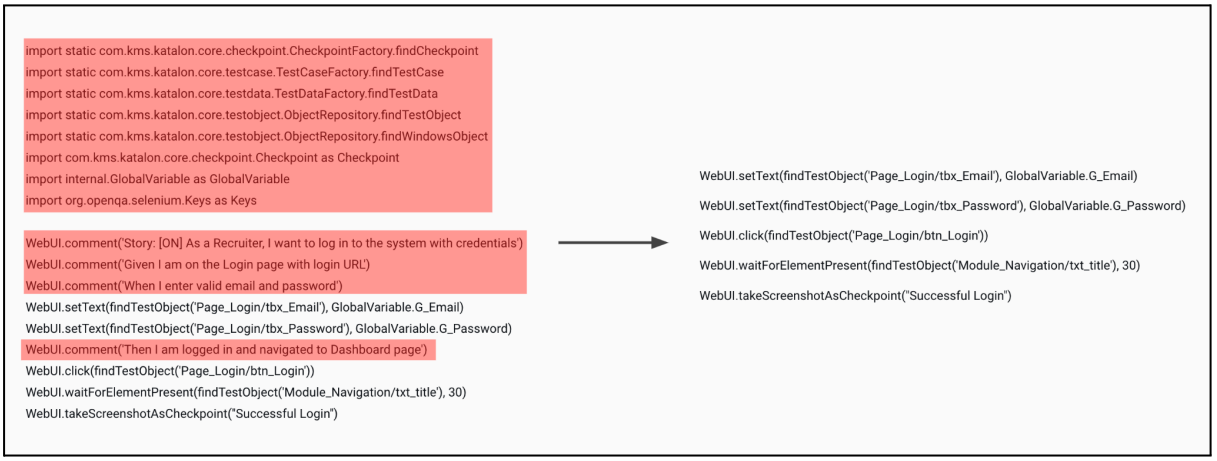
[One example of manual test case format]

On the other hand, for the test scripts, we collect relevant information like various function names, data types and object paths, a type of unique address referencing an element inside the software, like a login button, email textbox, etc., and store these in a JSON type format like a Python dictionary. All these files were further exported as pickle files that have been available in the project repository for exploration, use, and enhancements.

```
[{'action': 'callTestCase',
  'entities': ["findTestCase('Common Test Cases/Login/Login')",
   "[('Email') : GlobalVariable.G_Email, ('Password') : GlobalVariable.G_Password]",
   'FailureHandling.STOP_ON_FAILURE']},
 {'action': 'click',
  'entities': ["findTestObject('Object Repository/Page_UpdateProfile/img_Avatar')"]},
 {'action': 'click',
  'entities': ["findTestObject('Object Repository/Page_UpdateProfile/txt_MyProfile')"]},
 {'action': 'waitForElementPresent',
  'entities': ["findTestObject('Object Repository/Page_UpdateProfile/txt_email')",
   '20']},
 {'action': 'setText',
  'entities': ["findTestObject('Object Repository/Page_UpdateProfile/txt_FullName')",
   'newFullName']},
```

[One example of script format]

For the scripts, the focus was on removing lines of code that either were inconsequential, like comments, out of the scope of the current focus, like everything excluding the code corresponding to the main steps of the test case, and anything that couldn't be directly mapped, like the import statements, comments etc. Below is an illustration of how a simplified script is derived from an actual script, by removing import statements and comments. Moreover, as the import statements follow standard structure, we can simply prepend them to the generated script for the final output.



One characteristic of the test cases is that different test cases require different levels of complexity when translated into programming scripts. We have roughly divided the test cases into two main groups, one group includes the first 14 test cases where only a sequence of WebUI functions is required, whereas the other group, making up the rest of the test cases, requires more complex structures like loops and if statements. Due to the limitation of time, in this project, we focused solely on the first group of test cases.

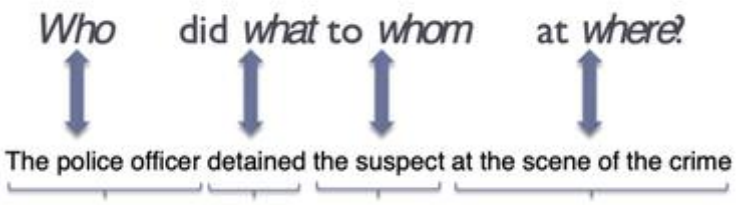
However, these extracted instructions are still not yet ready for actual processing. The reason for this is that many instructions are overly generalized and hard to be directly translated into code without real-world knowledge. For example, the instruction “Input an invalid credential to login”, which aims to test how the target program handles invalid credential input, is unclear about which steps are necessary to actually input and test invalid credentials. Abstract instructions like this pose great challenges to any translation pipeline without any prior domain knowledge. To handle this issue, we rewrote general instructions like the one above into more specific, executable steps, like “*Enter email address to Email textbox “invalid@wrong”*” and “*Enter password to Password textbox “invalidpassword”*”. Each of these steps corresponds to actual WebUI functions and contains information necessary for those functions. A more detailed list of how those frames are defined can be found in the appendix.

Translation pipeline

a) Parsing

With the newly rewritten test cases obtained as lists of instruction strings, we then parse the executable instructions. We begin the parsing process by first passing each instruction into an NLP package called AllenNLP to perform semantic role labeling (SRL). Semantic role labeling is the process of assigning labels to words

that indicate their semantic role in the sentence such as agent (who performs the action), and patient(who undergoes the action and changes its state). As illustrated in the figure below, a sentence like “the police officer detained the suspect at the scene of the crime” can be separated into four segments corresponding to “who” did “what” to “whom” at “where”, and semantic role labeling is the process of labeling these segment tags onto different parts of a sentence. Through some generalizable computational linguistic process, the AllenNLP module returns us the main verb of the instruction as well as objects in the sentence that directly relate to the main verb. From the SRL result, we extract the main verb as the action to be undertaken for this instruction. Notice that we are not utilizing other information extracted by AllenNLP here due to the limitation of the module. Even though AllenNLP is able to correctly extract the main verb of all instructions we have, it often fails to correctly extract other information possibly due to the fact that the model was trained on news data with little instruction sentence occurrences. For the manual instructions, lots of terms are capitalized to refer to the specific items in the software.



[a simplified version of semantic role labeling]

[One example of AllenNLP that fails to parse the semantic roles]

After successfully extracting the main verb from each instruction, we then create a corresponding frame defined for that verb with empty slot values. The frames contain slots that correspond to the different parameters the relevant programming test script function requires. For example, for action “input”, its corresponding function is “settext” that takes in a value to be entered and a location to enter the value as parameters. Therefore the frame for the verb “input” contains a value slot and a location slot.

Instead of AllenNLP, we train our own sequence tagger with manually annotated data from the rewritten test cases that results in 86 annotated instruction sentences for extracting named entity recognition like the value or text for the instructions. Sequence tagging can be understood as a problem where the model sees a

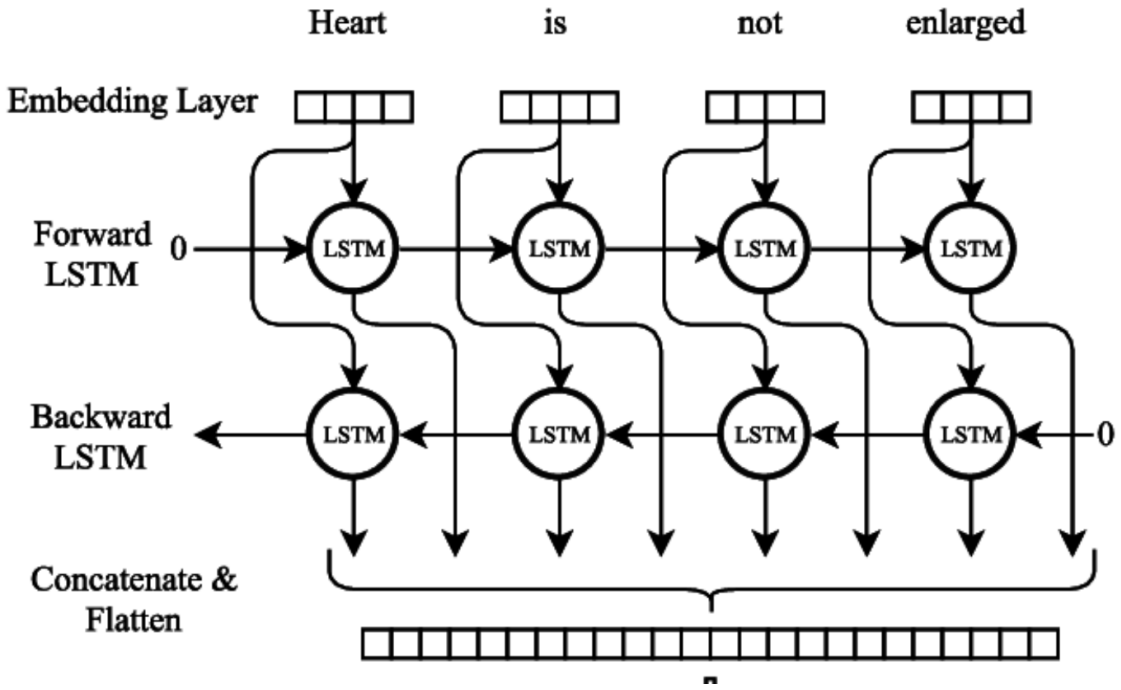
sequence of words or tokens for each word in the sequence and it is expected to emit a label. In other words, the model is expected to tag the whole sequence of tokens with an appropriate label from a known label dictionary.Our annotation framework is based on IOB format which means inside-outside-beginning. The I-prefix indicates that the tag is inside a chunk (i.e. a noun group, a verb group etc.). The O-prefix indicates that the token belongs to no chunk. The B-prefix indicates that the tag is at the beginning of a chunk that follows another chunk without O tags between the two chunks. [1] And we used BERT and BiLSTM with both stacked embeddings and Roberta Embeddings for training the tagger.

1	Word	IOB-tag
2	S1	*
3	Enter	O
4	email	O
5	address	O
6	to	O
7	Email	B-location
8	textbox	I-location
9	admin1@mail.cc	B-value
10	S2	*
11	Enter	O
12	password	O
13	to	O
14	Password	B-location
15	textbox	I-location
16	Admin@123	B-value
17	S3	*
18	Click	O
19	button	B-value
20	Login	I-value

[An example of IOB system]

BERT is an acronym for Bidirectional Encoder Representation from Transformers that consists of several transformer encoders stacked together and each encoder encapsulates two sub-layers: a self-attention layer and a feed-forward level. [2] The BERT model expects an input of a sequence of tokens/words and there two special tokens along with it: [CLS] refers to the first token of every sequence that represents classification token; [SEP] refers to a token that helps BERT to know sentence segmentation, which would be useful for next sentence prediction or question-answering task. A key note is that the maximum size of tokens to be fed into BERT is 512, and we could use [PAD] to fill the used token slots or use truncation if the tokens are longer than 512. For usual tasks like text classification, BERT only uses embedding vectors from the special [CLS] token while for named entity recognition tasks, we need to use the embedding vector output from all the tokens. [3] Another two models we used are based on Flair, one with stacked embeddings and another with Roberta as embedding. Flair is a NLP library that

builds on PyTorch that could perform tasks like named-entity recognition, parts-of-speech tagging, text classification. Flair is built on bidirectional LSTM, which is a sequence processing model consisting of two LSTMs: one takes the input in a forward direction and another tasks the input in backwards direction. So that the BiLSTMs could help to increase the amount of information available to the network and improve the context available for the algorithm. [4] The advantages of using Flair are that it comprises popular word embeddings such as GloVe, BERT, ELmo, character embedding, it also allows us to combine different word embeddings and it provides contextual string embeddings of Flair Embedding. [5] Embedding is essential for the breakthrough of deep learning on NLP tasks because each word is represented by a real-valued vector with tens or hundreds of dimensions that words with similar meaning would have a similar representation, whereas traditional one-hot encoding only has sparse word representations. [6] That is why a good embedding could better fulfill our tasks and generalize the model more based on the similarity of the words and context.



[shows a BiLSTM model with embedding layer]

In total, three neural taggers were trained for NER tagging and were evaluated using F1-score of the predicted tags w.r.t the manual annotated tags.

$F1\ score\ = \frac{2 * precision * recall}{(precision + recall)}$.

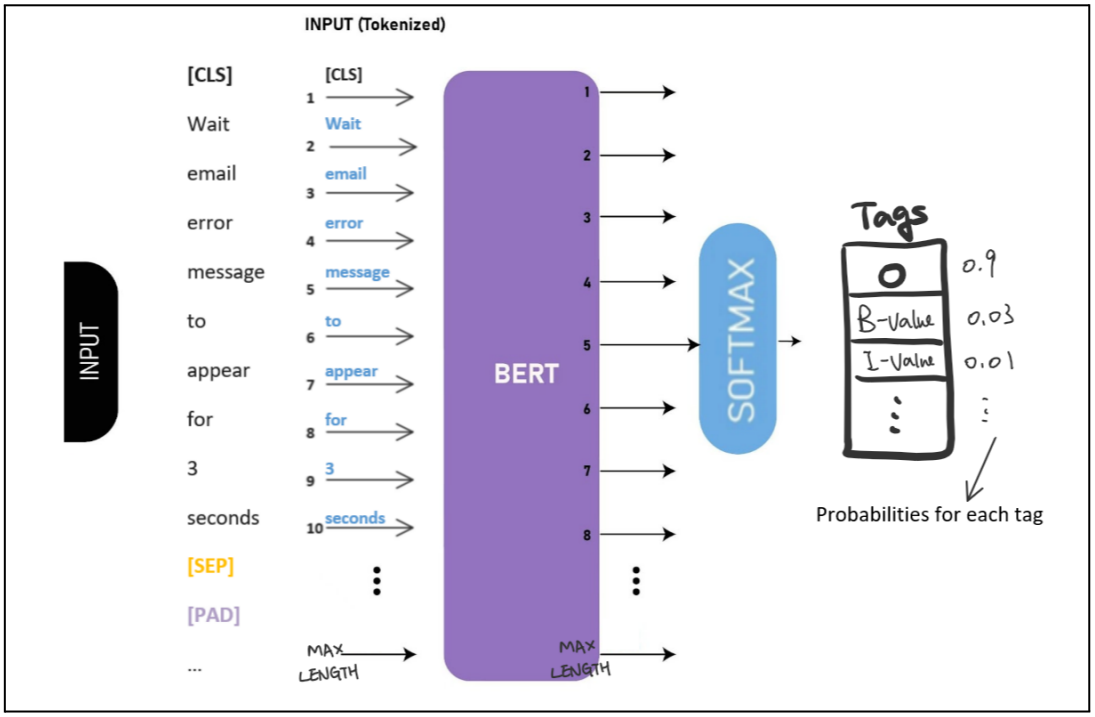
```
Click Change Password link
Predict:
['O', 'B-value', 'B-value', 'I-value']
Gold:
['O', 'B-value', 'I-value', 'I-value']
```

[An example of predicted tags vs. manual annotated tags]

```
Precision  = 3 / 4 = 0.75
Recall     = 3 / 3 = 1
F1 -score  = 2 * 0.75 * 1 / (0.75 + 1) = 0.8571
```

[F1-score calculation based on example above]

BERT and Flair's BiLSTM models both achieved around 95% F1 score and both above 90% recall and precision. However, since LSTM models could achieve higher results than a BERT model for a small dataset [7], once the Katalon project adds more datasets in the future and trains for the tagging, BERT could potentially perform better than the LSTM model. Therefore, we'll use the BERT tagger for the task. The result of the BERT tagger on the dev set (20% of the manual annotated test steps from the first 14 rewritten test cases) is – Precision: 98.18%, Recall: 100.00%, F-score: 99.08%.



[Shows how BERT tagger works]

b) Script Generation

Once the test cases have been pre-processed, they would be used by the script processing workflow in a dictionary-like format. It includes the parsed verb and the associated entities.

For the next step, the processed test steps will be aligned to a WebUI function with arguments. A dictionary with WebUI function names as keys and sets of corresponding action verbs as values was built for action verbs and WebUI function names alignment.

```
{'setText': {'enter', 'input'},
 'click': {'click'},
 'waitForElementPresent': {'wait'},
 'getText': {'get'},
```

[Part of the WebUI function - Action verb dictionary]

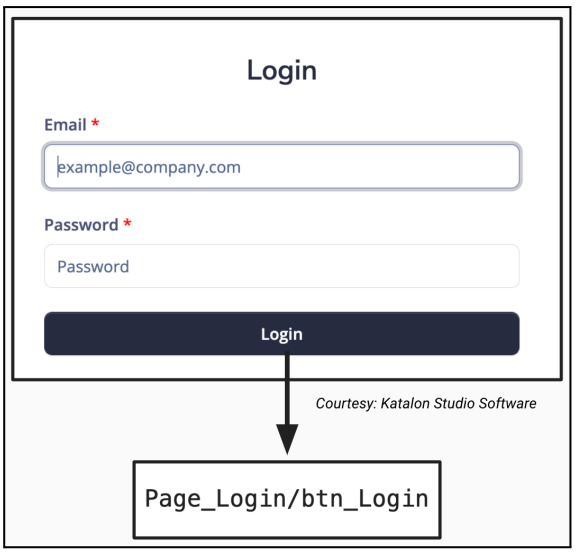
In case an input action verb is not included in this dictionary, SpaCy(cosine similarity of embedding vectors) was used to calculate word similarity between the input action verb and each action verb in the dictionary, then the WebUI function that corresponds to the most similar verb in the dictionary will be chosen as the aligned WebUI function.

With the aligned WebUI function, we use a rule-based approach to get the arguments needed for this WebUI function. Since each WebUI function requires different input arguments.

Finding object path

a) What is an object path?

An object path is a unique address or an identifier for a software element. Since, the process of accessing a software is automated, for a script to interact with an element inside the software, it should be able to uniquely identify it. An object path helps achieve that purpose. It consists of the page Information as well as the element name.



[The object path for the login button inside the Katalon Testing Software]

b) Our approach

Data Structures used

As briefly discussed in the Data Preprocessing section of the report, we collected all the object paths from the scripts and stored it as a set, referred to as a path set. We further segregated the paths based on the paths' page information and element name, and stored the information in two types of dictionaries:

1. path_dict_c: It consists of page information, their corresponding elements, and the corresponding object paths. Below is an example of what the path_dict_c looks like for the page key: 'Page_Login'.

```
'Page_Login': defaultdict(set,
            {'Logout': {'Page_Login/btn_Logout'},
             'Password': {'Page_Login/tbx_Password'},
             'Login': {'Page_Login/btn_Login'},
             'PasswordErrorMessage': {'Page_Login/txt_PasswordErrorMessage'},
             'EmailErrorMessage': {'Page_Login/txt_EmailErrorMessage'},
             'Email': {'Page_Login/tbx_Email'},
             'ChangePassword': {'Module_Navigation/btn_ChangePassword'},
             'title': {'Module_Navigation/txt_title'},
             'Members': {'Module_Navigation/nav_Members'},
             'Avatar': {'Module_Navigation/lbl_Avatar'},
             'breadcrumb': {'Module_Navigation/breadcrumb'},
             'item_libraries': {'Module_Navigation/nav_item_libraries'},
             'Integrations': {'Module_Navigation/nav_Integrations'},
             'Libraries': {'Module_Navigation/btn_Libraries'},
             'item_tests': {'Module_Navigation/nav_item_tests'},
             'Success': {'Module_Navigation/toast_Success'}}),
```

2. alternate_dict_c: It consists of elements, their corresponding pages, and the corresponding object paths. Below is an example of what the alternate_dict_c looks like for different 'element' keys and their corresponding page keys.

```
{'search button': defaultdict(set,
             {'Candidates': {'Page_Candidates/btn_SearchButton'}}),
 'informative text': defaultdict(set,
             {'Candidates': {'Page_Candidates/txt_InformativeText'}}),
 'table row': defaultdict(set,
             {'Candidates': {'Page_Candidates/tr_TableRow'}}),
 'bulk cancel': defaultdict(set,
             {'Candidates': {'Page_Candidates/btn_bulkCancel'},
              'Test Invited': {'Page_TestInvited/btn_Bulk Cancel'}}),
 'scored candidate': defaultdict(set,
             {'Candidates': {'Page_Candidates/btn_ScoredCandidate'}}),
 'email status': defaultdict(set,
             {'Candidates': {'Page_Candidates/select_EmailStatus'}}),
 'candidate title': defaultdict(set,
             {'Candidates': {'Page_Candidates/txt_CandidateTitle'}}),
 'bulk review': defaultdict(set,
             {'Candidates': {'Page_Candidates/btn_bulkReview'}}),
 'headers': defaultdict(set,
             {'Candidates': {'Page_Candidates/th_Headers'}}),
```

Furthermore, to know which object path has been historically used inside which function, a mapping between the function name and a set of associated object paths was also accumulated and stored in a Python dictionary, termed as action_widget. Later, some of the function names were changed to match with the action verb identified during the parsing workflow. For example, ‘setText’ function was changed to two keys, ‘enter’ and ‘input’ to align the terms with the ones established in the parsed representations of the test case. Below is an example of the action_widget dictionary for the action verb ‘enter’.

```
'enter': {'Page_Candidates/tbx_SearchCandidate',
 'Page_ChangePassword/tbx_ConfirmPassword',
 'Page_ChangePassword/tbx_CurrentPassword',
 'Page_ChangePassword/tbx_NewPassword',
 'Page_Create Member/inp_Email',
 'Page_Create Member/inp_Full Name',
 'Page_EmailTemplates/ipn_Subjective',
 'Page_Libraries/Library_Dialog/tbx_LibraryName',
 'Page_Libraries/Library_Dialog/txt_nameField',
 'Page_Libraries/Question_Dialog/tbx_ChoiceValue',
 'Page_Libraries/Question_Dialog/tbx_MaxTime',
 'Page_Libraries/Question_Dialog/tbx_Tags',
 'Page_Libraries/tbx_SearchQuestion',
 'Page_Login/tbx_Email',
 'Page_Login/tbx_Password',
 'Page_PreviewTest/textarea_Feedback',
 'Page_SearchTests/txt_search',
 'Page_SendFeedback/textarea_Feedback',
 'Page_Set Password/txt_Password',
 'Page_Set Password/txt_Repassword',
 'Page_TestCandidates/input_SearchCandidate',
 'Page_TestOverview/input_TestTags',
 'Page_TestOverview/input_TextOrNumberField',
 'Page_TestQuestions/tbx_AwardedScore',
 'Page_TestQuestions/tbx_MaxTime',
 'Page_TestQuestions/tbx_SubtractedScore',
 'Page_UpdateProfile/txt_FullName',
 'Page_UpdateProfile/txt_Phone'},
```

Strategy

The idea is to compare the information accumulated through parsing with the page information and the element name of the object path and zero down on the possible object path or a number of object paths corresponding to that information. The eventual execution of the strategy was divided into three sub-strategies, defined as follows:

Primary strategy

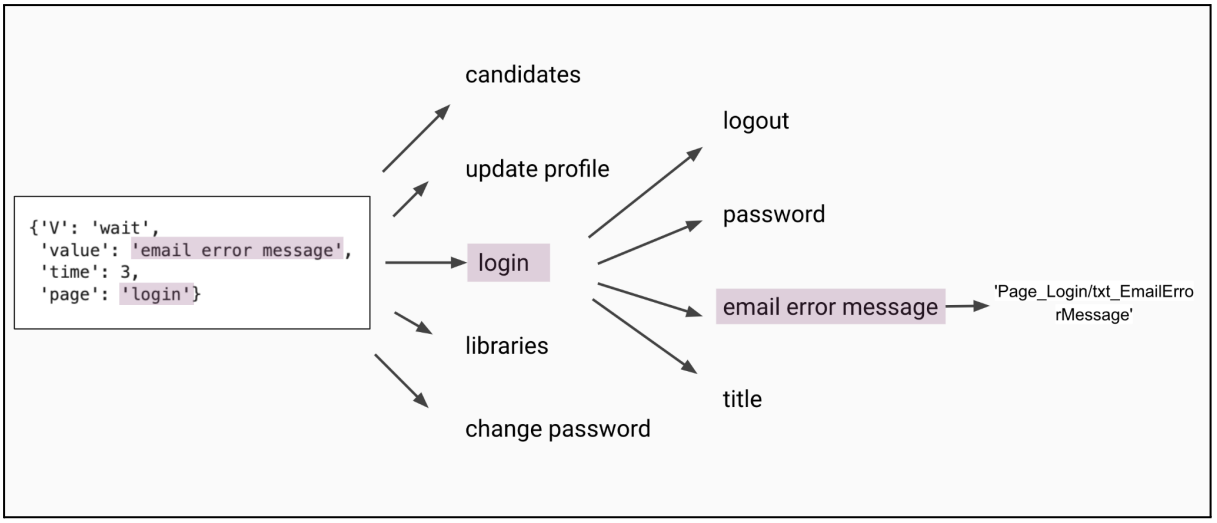
The primary strategy makes use of the path_dict_c and the action_widget dictionary.

- From the parsed representation of the test case, it first compares the ‘page’ property with the ‘page’ keys of the path_dict_c data structure and chooses the most semantically similar page key.
- Then it compares the ‘location’ property or the ‘value’ property (if ‘location’ is not available) with the element keys inside the identified page and selects the one most string-wise similar to the

aforementioned properties. Here string-wise refers to the similarity of the characters used. For example, ‘login’ is more similar string-wise to ‘logout’ than ‘sign in’, because more characters are similar (‘log’ in both) in the former case.

- Once it has identified an object path or set of object paths in this manner, it verifies whether the action verb (‘V’) of the parsed test case contains the zeroed down object paths in the action_widget dictionary.
- All the identified paths that also correspond to the action verb in the action_widget dictionary are then returned.

Primary strategy works for most of the cases since it’s easier to establish the semantic similarity of the pages first and then work through its corresponding elements. However, in some situations, it is possible that the page information is not correctly identified, and this is where the alternative strategy comes in.

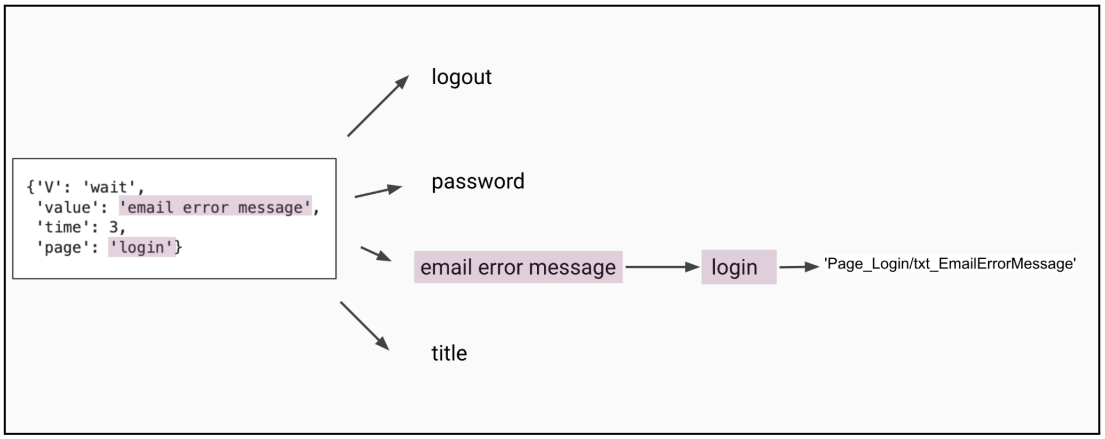


Fallback Strategy

The fallback strategy makes use of the alternative_dict_c and the action_widget dictionary.

- From the parsed representation of the test case, it first compares the ‘location’ / ’value’ property with the ‘element’ keys of the alternate_dict_c data structure and chooses the most string-wise similar page key.
- Then it compares the ‘page’ property with the ‘page’ keys of the path_dict_c and identifies the most semantically similar page key.
- Once it has identified an object path or set of object paths in this manner, it verifies whether the action verb (‘V’) of the parsed test case contains the zeroed down object paths in the action_widget dictionary.
- All the identified paths that also correspond to the action verb in the action_widget dictionary are then returned.

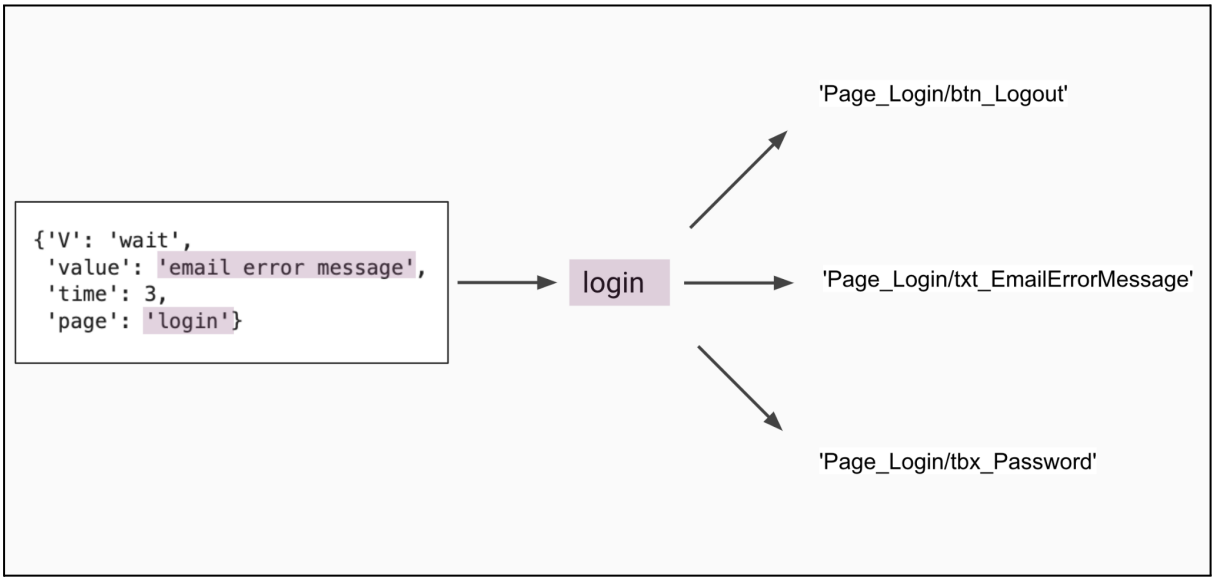
In a few scenarios, where the fallback strategy is actually implemented, it does the job almost every time. However, in an unlikely scenario, where even the alternative strategy fails to return an object path, specifically in cases where different elements are string-wise too similar, we implement the last course of action, fittingly named ‘The last resort’.



The last resort

The last resort makes a partial use of the path_dict_c dictionary as well as the action_widget dictionary.

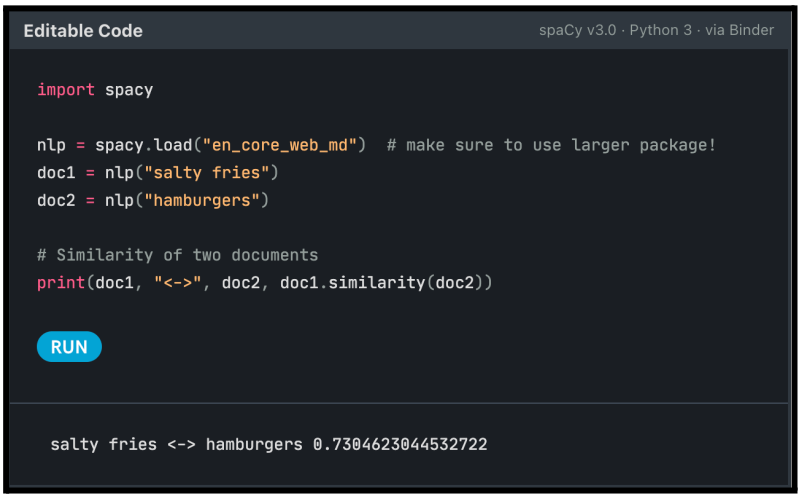
- From the parsed representation of the test case, it first compares the ‘page’ property with the ‘page’ keys of the path_dict_c data structure and chooses the most semantically similar page key.
- Further, it accumulates all the object paths associated with that page key, without zeroing down on a particular element like in the case of the primary strategy.
- After accumulating these object paths, it verifies whether the action verb (‘V’) of the parsed test case contains the zeroed down object paths in the action_widget dictionary.
- All the identified paths that also correspond to the action verb in the action_widget dictionary are then returned.



String comparison metrics

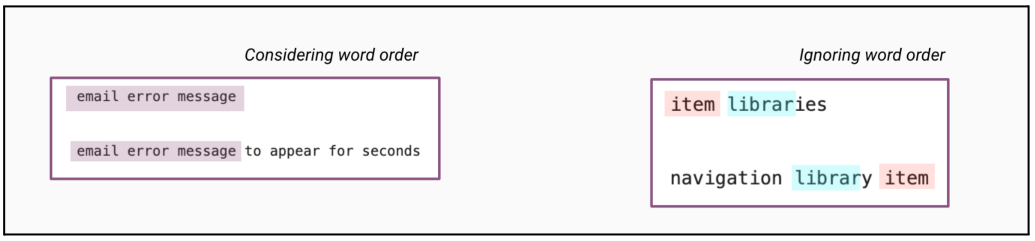
To compare aspects of the object path with the parsed test case, we used three kinds of metrics, namely:

- Semantic similarity: It vectorizes the strings in a multi-dimensional vector space and measures the cosine similarity between the two. Specifically for this project, semantic similarity is used to identify the ‘page’ key corresponding to the page property of a parsed test-case for all the three strategies. The range of semantic similarity is between 0 and 1, with 1 being the most similar and 0 being the least.

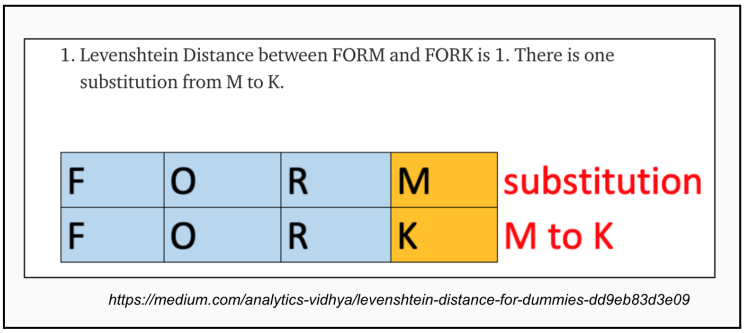


https://spacy.io/usage/linguistic-features#vectors-similarity

- Longest common substring: Measures the size of the longest common substring between two strings. It has been used in two separate ways. For the primary strategy, the longest common substring is calculated for two words between the two strings at a time and summed up instead of measuring the longest substring between the entire strings at once. This helps to ignore the word order between the element key of the path_dict_c and ‘location’ / ‘value’ property of the parsed test-cases. It helps because there are less number of elements to be compared in the primary strategy as only those elements associated with the earlier identified page are up for contention. So it helps to make up for the cases where the word order between the two strings is different. However, for the alternate strategy we compare entire strings since the number of elements here are more in number and the word order becomes more important to distinguish between different cases that could otherwise be quite similar. The longer the common substring between two strings, the more similar they’re.



- Levenshtein Distance: Quantifies the effort required to make two strings similar. The smaller the levenshtein distance, the more similar the two strings are. Levenshtein distance is used to identify the ‘element’ key corresponding to the ‘location’/ ’value’ property when the longest common substring scores are the same for both the strings.



Quantifying the comparison

While the semantic similarity and levenshtein distance are used as such, the score for the longest common substring is evaluated differently. It is also different for the two strategies, given as follows:

- Primary strategy: (Length(‘location’/ ’value’ property) - longest common substring)/Length(‘location’/ ’value’ property)
- Alternate strategy: (Length(‘element’ key) - longest common substring)/Length(‘element’ key)

Another thing to note when it comes to comparisons is that the strings being compared are processed using regular expressions before the comparison. For example: ‘Page_Login’ is processed as ‘login’ and ‘EmailErrorMessage’ is processed as ‘email error message’.

Results

1. Out of 61 total object paths, 54 were identified correctly.
2. For the remaining 7, a set of 2-3 object paths was returned where the correct path in the returned set was included for all the 7 cases.

Based on the aforementioned numbers, it is safe to say that the results have largely been favorable.

After finding the object path for needed arguments, we use a rule-based approach to reformat the dictionary format WebUI function with arguments into the script in string.

The Script processing workflow includes a lot of rules to generate corresponding functions, variables, and function arguments and relies heavily on the data structures constructed from the provided script data and object path repository to find object paths, an entity that uniquely identifies various elements of the software interface to be tested in this case. Both the rules and the data structures are being continuously improved to suit newer cases that help inform the scope of generalization and what strategies could help us achieve that.

Evaluation Methods

In detail, after we have the generated code script, we also load the script to create the gold standard. And then, we utilized a self-defined parser to parse the Groovy into smaller chunks and compare the scripts on script level and average the score by the number of scripts.

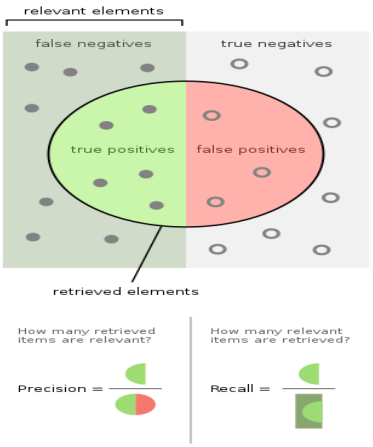
```
script1
```

```
["WebUI.setText(findTestObject('Page_Login/tbx_Email'), 'invalid@wrong')",
 "WebUI.setText(findTestObject('Page_Login/tbx_Password'), 'invalidpassword')",
 "WebUI.click(findTestObject('Page_Login/btn_Login'))",
 "WebUI.waitForElementPresent(findTestObject('Page_Login/txt_EmailErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)",
 "def actualEmailErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_EmailErrorMessage'))",
 "WebUI.waitForElementPresent(findTestObject('Page_Login/txt_PasswordErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)",
 "def actualPasswordErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_PasswordErrorMessage'))"]
```

```
groovy_parser(script1[0])
```

```
['WebUI.setText',
 'findTestObject',
 "'Page_Login",
 "tbx_Email'",
 " 'invalid@wrong'"]
```

1. $F1\ score\ = \frac{2 * precision * recall}{(precision + recall)}$



F1 one is a popular metric for evaluating the performance of classification models. The formula is shown above. Recall means of all the actual positive examples out there, how many of them did I correctly predict to be positive and it is calculated by the correct generated code chunks divided by (correct generated code chunks + code chunks not correctly generated=false negatives). And precision means of all the positive predictions I made, how many of them are truly positive. It is calculated by the correctly generated code chunks divided by all the generated code chunks (both right and wrong ones). F1 is one of the widely used metrics because it considers both precision and recall and calculates a balanced summarization of the model performance.

However, since F1 does not care about the order of the generated elements, we are also using BLEU score as another metric, which cares about the order of the generated elements by comparing the 1-4 grams match.

2. BLEU score

BLEU (BiLingual Evaluation Understudy) is a metric for automatically evaluating machine-translated text. The BLEU score is a number between zero and one that measures the similarity of the machine-translated text to a set of high-quality reference translations. [9] It is calculated by using n-grams precisions and brevity-penalty.

BLEU

- N-gram overlap between machine translation output and reference translation
- Compute precision for n-grams of size 1 to 4
- Add brevity penalty (for too short translations)

$$\text{BLEU} = \min\left(1, \frac{\textit{output-length}}{\textit{reference-length}}\right) \left(\prod_{i=1}^{4} \textit{precision}_i\right)^{\frac{1}{4}}$$

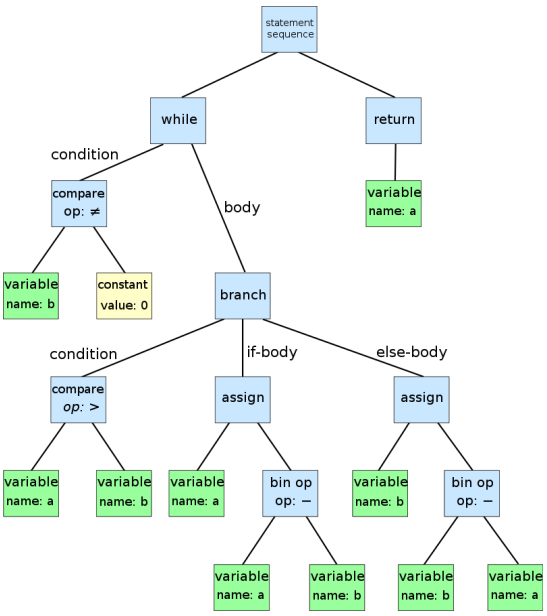
- Typically computed over the entire corpus, not single sentences

- Brevity Penalty
  The brevity penalty penalizes generated translations that are too short compared to the closest reference length with exponential decay. The brevity penalty compensates for the fact that the BLEU score has no recall term.[10]
- N-Gram Overlap
  The n-gram overlap counts how many unigrams, bigrams, trigrams, and four-grams (i=1,...,4) match their n-gram counterpart in the reference translations. This term acts as a precision metric.[11] Unigrams account for adequacy while longer n-grams account for fluency of the translation. To avoid overcounting, the n-gram counts are clipped to the maximal n-gram count occurring in the reference

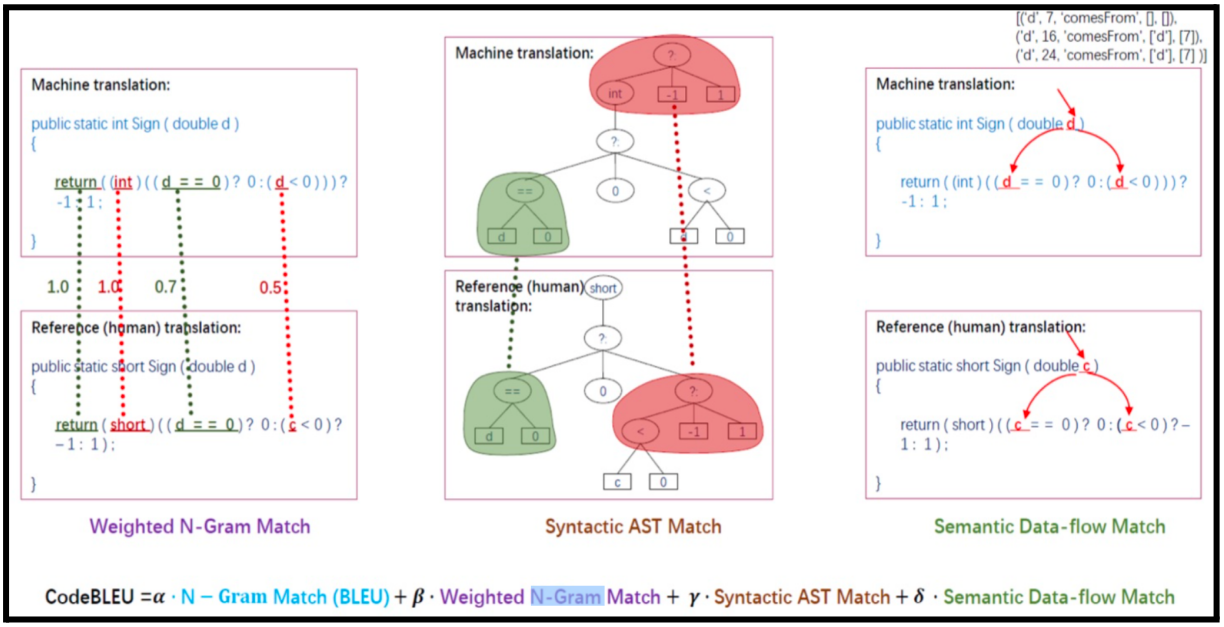
For code, we import the sentence_bleu from nltk and use a smoothing function method 1 to add epsilon count to precision with 0 counts to avoid zero division and calculate n-grams(1-4) BLEU score with equal weights.

3. CodeBLEU

Originally we proposed to use CodeBLEU[8] as an evaluation method because compared to BLEU score, it takes the syntax and semantics of the translation into consideration. In detail, Code script has abstract syntactic trees that represent the structure of the programming language, with nodes denoting a construct occurring in the source code and leaves presenting the names of the functions or all the variables. By using the AST trees, we would give more weight to the syntactic structure of the nodes and less to the naming variables. And it also has semantic data flow that shows that different variables mean the same thing in different translations. For instance, car and automobile have the same meaning in most of the cases in our language and should be treated as interchangeables and same as when we use different variable names in code translation like use A or C for variable names shown in the graph.



[An example of AST trees, the nodes means construct of the function and leaves as the variables]



However, it needs a special parser for Groovy so that it could parse the code into tree structure like natural language parsers and it also needs to identify the semantic flow in the code which requires specialized packages developed for Groovy. Currently, the packages are only available for languages like Java, Python and though Groovy is similar to Java, it could not parse it. Plus, we also need to determine the specific weight for the n-gram match for the particular language and the weights of syntactic AST match and semantic data-flow match.

CodeBLEU is computed as an average(or with other weights) of the three measures and BLEU is one of those. If there are available packages in the future for Groovy language syntactic tree parsing, CodeBLEU would be a much better metric.

4. Runs perfectly (Yes/No)

Initially, we planned to use the metrics of whether it runs on Katalon Studio as one of the evaluation metrics. However, the later steps of the code scripts, it has lots of preconditions and verifications and we only did the main test cases. Without the pre-condition script in the code, it would not be able to accomplish the goal. Therefore, we are not using it as a metric.

5. Results

Outputs of the pipeline

An example generated test script

```
WebUI.setText(findTestObject('Page_Login/tbx_Email'), 'invalid@wrong')
WebUI.setText(findTestObject('Page_Login/tbx_Password'), 'invalidpassword')
WebUI.click(findTestObject('Page_Login/btn_Login'))
WebUI.waitForElementPresent(findTestObject('Page_Login/txt_EmailErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)
def actualEmailErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_EmailErrorMessage'))
WebUI.verifyMatch(actualEmailErrorMessage, expectedEmailErrorMessage, false)
WebUI.waitForElementPresent(findTestObject('Page_Login/txt_PasswordErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)
def actualPasswordErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_PasswordErrorMessage'))
WebUI.verifyMatch(actualPasswordErrorMessage, expectedPasswordErrorMessage, false)
```

According to this rewritten test case

```
Steps:
1. Enter email address to Email textbox “invalid@wrong“
2. Enter password to Password textbox “invalidpassword”
3. Click [Login] button
4. Wait for email error message to appear
5. Get email error message
6. Check if actual error message is correct
7. Wait for password error message to appear
8. Get password error message
9. Check if actual password message is correct
```

Evaluation results of translation (F1 Score)

```python
def f1_score(reference, test):
    correct = 0
    for chunk in reference:
        if chunk in test:
            correct += 1
    recall = correct / len(reference)
    precision = correct /len(test)
    f1 = 2 * recall * precision / (recall + precision)
    return f1
```

(note: script6 is empty so we ignored it)

We achieved 0.642 for the F1 score for the first 14 cases if based on purely rule-based methods.
If using tagging to extract the entities, the F1 score is 0.603 because it fails to generate one script.

Evaluation results of translation (BLEU)

For the first 14 cases, we have 47 for 1-gram BLEU score, 35 for 2-gram BLEU score, 29 for 3-gram BLEU score, 25 for 4-gram BLEU score by using purely rule-based methods.

By using tagging to extract entities, we have 37 for 1-gram BLEU score, 29 for 2-gram BLEU score, 23 for 3-gram BLEU score, 20 for 4-gram BLEU score.

BLEU Score	Interpretation
< 10	Almost useless
10 - 19	Hard to get the gist
20 - 29	The gist is clear, but has significant grammatical errors
30 - 40	Understandable to good translations
40 - 50	High quality translations
50 - 60	Very high quality, adequate, and fluent translations
> 60	Quality often better than human

One thing to note is that in certain cases, we generated a correct but different variable name for some of the code chunks. While using BLEU, a different variable would simply be marked as incorrect and that decreased the BLEU score significantly.

For instance, In script1, the gold script uses email and password as variable names while our script uses specific emails. However, they mean the same thing to illustrate that the email is entered wrong. This is the limitation of using the BLEU score as it is more suited for natural language translation evaluation. (CodeBLEU could possibly be better because it considers the semantic data flow consideration which will treat variables with different names as correct or give partial credit. Though we are not able to implement CodeBLEU in this project, it might be useful once there are more resources and packages available in the future.

```
["WebUI.setText(findTestObject('Page_Login/tbx_Email'), Email)",
 "WebUI.setText(findTestObject('Page_Login/tbx_Password'), Password)",
 "WebUI.click(findTestObject('Page_Login/btn_Login'))",
"WebUI.setText(findTestObject('Page_Login/tbx_Email'), 'invalid@wrong')",
"WebUI.setText(findTestObject('Page_Login/tbx_Password'), 'invalidpassword')",
"WebUI.click(findTestObject('Page_Login/btn_Login'))",
```

6. Analysis

Parsing Analysis

Most of the errors that are caused by the BERT tagger are that some words are separated into several parts after tokenization and cannot be combined back together. Here is an example. *“Invalid@wrong”* was tokenized into 5 parts: *“, invalid, @, wrong,* and *”.* After decoding, white spaces were left between those parts. Consequently, the extracted values do not match with the target value exactly.

```
Test step:
Enter email address to Email textbox "invalid@wrong"
Tokens:
['[CLS]', 'En', '##ter', 'email', 'address', 'to', 'Em', '##ail', 'text', '##box', '"', 'invalid', '@', 'wrong', '"', '[SEP]']
Text after combing tokens:
Enter email address to Email textbox " invalid @ wrong "
Predict:
['O', 'O', 'O', 'O', 'B-location', 'I-location', 'O', 'B-value', 'I-value', 'I-value', 'B-value']
Gold:
['O', 'O', 'O', 'O', 'B-location', 'I-location', 'B-value']
```

On the other hand, there is no error when we use AllenNLP to parse the main verb from each instruction sentence. The reason for this high performance is that our instruction sentences have a clear and consistent structure where the main verb is always placed at the first position of the sentence. Therefore for a generalizable model like AllenNLP, correctly extracting the main verb is a very simple task.

Script Generation Analysis

The alignment consists of two parts, action verb – WebUI function alignment, and arguments extraction. When an input action verb is unknown, word similarity calculation will be used to determine the aligned WebUI function. We did not get a chance to test this SpaCy similarity calculator. This part might cause some errors. For argument extraction, it is mainly rule-based since different arguments are required for different functions. Therefore, the limited rules are a problem here. If the function is not covered by the rules, it is highly possible that the generated script line is wrong. As shown in the example below, the *takeScreenshotAsCheckpoint* function was not generated since it was not included in the rules.

```
WebUI.setText(findTestObject('Page_Login/tbx_Email'), Email)
WebUI.setText(findTestObject('Page_Login/tbx_Password'), Password)
WebUI.click(findTestObject('Page_Login/btn_Login'))
WebUI.takeScreenshotAsCheckpoint("Failed Login")
WebUI.waitForElementPresent(findTestObject('Page_Login/txt_EmailErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)
def actualEmailErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_EmailErrorMessage'))
WebUI.verifyMatch(actualEmailErrorMessage, expectedEmailErrorMessage, false)
WebUI.waitForElementPresent(findTestObject('Page_Login/txt_PasswordErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)
def actualPasswordErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_PasswordErrorMessage'))
WebUI.verifyMatch(actualPasswordErrorMessage, expectedPasswordErrorMessage, false)
```

[Given test script]

```
WebUI.setText(findTestObject('Page_Login/tbx_Email'), 'invalid@wrong')
WebUI.setText(findTestObject('Page_Login/tbx_Password'), 'invalidpassword')
WebUI.click(findTestObject('Page_Login/btn_Login'))
WebUI.waitForElementPresent(findTestObject('Page_Login/txt_EmailErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)
def actualEmailErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_EmailErrorMessage'))
WebUI.verifyMatch(actualEmailErrorMessage, expectedEmailErrorMessage, false)
WebUI.waitForElementPresent(findTestObject('Page_Login/txt_PasswordErrorMessage'), 3, FailureHandling.STOP_ON_FAILURE)
def actualPasswordErrorMessage = WebUI.getText(findTestObject('Page_Login/txt_PasswordErrorMessage'))
WebUI.verifyMatch(actualPasswordErrorMessage, expectedPasswordErrorMessage, false)
```

[Generated script]

[One example of Given test script vs Generated script]

As far as finding the object path goes, there's no identifiable error of sorts but the difference in how the longest common substring scores are calculated differently for the two strategies without any reasonable explanation is indeed a point that needs to be highlighted and discussed. Calculating the scores differently definitely helps for the limited number of test cases tested, however, how it might generalize with more cases to be tested and what role the separate scoring strategies plays remains to be seen. Furthermore, the data collected for the action_widget dictionary has missed some of the function -> object path mapping which were eventually added manually.

7. Future work

Parsing

In the next steps, the parsing modules can be extended in several ways. The first possible extension is that we could define more frames for more instructions with new main verbs, for example, 'verify'. This increases the translation pipeline's ability to handle more kinds of instructions and is the easiest extension to achieve.

We also propose two more complex extensions to our parsing module. At the moment, our translation pipeline solely focuses on the main steps in each test case. One useful addition to the parser is some mechanisms to parse the pre-conditions as well. In particular, some of the later test cases require rerunning previous test case scripts and those requirements are specified in the pre-condition section. Besides, the parser module can also extend to process test cases that require more complex test scripts. For example, some test cases require for loops or if statements in the generated scripts. It remains a question to our pipeline as to how to process the test cases in order to produce such code structures.

For the tagging to extract named entity recognition, more datasets are needed to train the tagger to generalize. Therefore, it would be important to collect more steps and also consider how the users would actually write the instructions in English. After obtaining enough datasets, IOB format annotation could be useful to build the training and validation dataset. BERT model is recommended for the training

because it is the state-of-art technique for named entity recognition and it is fast to train and fine-tune.

The tokenization issue of BERT tagger that mentioned in the Parsing Analysis part also needs to be taken care of in the future. For now, we combine all labeled values with white spaces in between. For example, if the instruction is “click on the submit button” and the target value is “submit” and “button”, we extract the value as “submit button”. However, this means that in cases where the BERT tokenizer incorrectly breaks value down into parts. For example, *“Invalid@wrong”* was tokenized into 5 parts: “, *invalid, @, wrong,* and ”. It would be extracted as ““ Invalid @ wrong ””. Maybe try adding another step to identify these special cases and put the pieces back together before doing the slot filling.

Script Generation

Function - action verb alignment

WebUI function-action verb dictionary

It is included in `script_generation.py`. Please keep updating the dictionary to adapt the new pairs of WebUI function name and action verb.

Similarity calculation

For the situations where an action verb is not included in the dictionary above, currently we use Spacy to calculate word similarity and choose the WebUI function that corresponds to the most similar action verb in the dictionary.

Maybe try different word similarity metrics in the future, such as longest common substring.

It is the `align_func` function in `script_generation.py` file.

Get object path

The `get_object_path` module returns one or a set of object paths in string format. When it returns more than one path, currently we choose a random path from the output set. Ideally, we’d like to rope in the user to identify the correct object path from the set returned, however the user interaction feature for the same has not been established yet in the code generation workflow.

Another factor to consider for the future could be to look into the separate scoring strategies for the longest common substring methodology and how well it generalizes. In case it doesn't, which of the two strategies could work better or if a different strategy altogether needs to be considered are some of the questions that ought to be discussed. In general, it needs to be looked into whether the methodologies implemented in the get_object_path module needs to be uniformalized and how the same could be achieved.

Also, the action_widget dictionary, itself derived from another data structure, namely the func_obj_dict, ought to be reworked to include some of the cases for which the association between the function and the object path was not established.

It is the `obj_path` function in the `script_generation.py` file.

Rule-based code generation

The rules are depending on the action verbs. We only considered very limited action verbs for now. The rules need to be kept updated to accommodate new action verbs.

8. Conclusion

We build a pipeline that can automatically translate manual test user-cases in English into Groovy script that helps accelerate the process of automation testing by eliminating the need to manually write the programming script for the same. The pipeline involves rewriting the test cases, frame parsing by neural network-based AllenNLP packages, and training named entity recognition tagger for extracting entities of interest for slot filling; aligning function name with slot values, finding object path using various string comparison metrics: Levenshtein distance, word vectors, and semantic similarity, Longest common substring, and finally generating the script. We used F1 and BLEU scores to evaluate the generated code script and achieved 0.68 F1-score and 30 for BLEU score.

Due to limited data and time, there are several aspects we could work on to improve the pipeline and build models with higher F1 and BLEU scores with more generalizability. To increase the accuracy and generalizability of the translation part of the pipeline system, future work could involve focussing on extending the current framework to cover more instructions with simpler structures, training tagger to parse and extract entities of interest, and also handle pre-conditions. And for the script generation part, the focus could be action verb alignment, getting object path, and generating rule-based code.

In conclusion, we delivered a combination of generalizable machine learning and rule-based models. The pipeline could process the test case instructions and handle different variations, extracting the action and tagging relevant entities. These combinations offer us a prototype or framework that the product could be built upon for code generation from natural language for software testing.

9. Schedule

Week 1 (May4-10): Understand task & data

Zhiyi worked on researching evaluation metrics specific to our programming script translation task;

Anshul and Jeremy worked on extracting and aligning manual test cases and programming scripts;

Yujie worked on extracting a list of all possible test objects.

Week 2 - 3 (May11-17): Process data and annotate test cases

Zhiyi & Jeremy worked on processing steps in manual test cases;

Anshul & Yujie worked on processing programming scripts.

Week 3 (May18-24): Keep working on preprocessing and make alignment between manual test cases and test scripts.

Zhiyi & Jeremy worked on parsing manual test cases;

Anshul & Yujie worked on processing programming scripts;

Everyone worked on making alignment.

Week 4 (May25-27): Complete and evaluate the pipeline for the first 8 pairs of test cases and test scripts

Jeremy worked on rewriting manual test cases;

Zhiyi worked on evaluation for the pipeline;

Yujie worked on generating scripts;

Anshul worked on finding the object paths.

Week 5 (Jun1-7): NER annotations for rewritten first 8 cases and created SpaCy tagger for parsing

Everyone worked on completing annotation;

Zhiyi & Yujie worked on the Spacy tagger;

Jeremy kept working on parsing;

Anshul kept working on improving finding the object paths.

Week 6 (Jun 8-14): More annotations and different taggers

Jeremy & Anshul worked on more test cases – rewriting and parsing;

Zhiyi & Yujie tried out more taggers;

Everyone worked on annotations.

Week 7 (Jun15-21): Try out taggers with more annotations + Clean up code + Evaluation on final pipeline

Zhiyi & **Jeremy** should be responsible for cleaning up code related to manual test cases processing;

Anshul & **Yujie** should be responsible for cleaning up code related to programming test script processing;

Zhiyi & **Yujie** should be responsible for cleaning up the code for taggers.

Week 8 (Jun22-29): Final report and video presentation

This would be equally split amongst team members according to the parts that each person was specifically responsible for.

10. Appendix

Project Repository

https://github.ubc.ca/mds-cl-2021-22/katalon_project/tree/master

References

1. Building named entity recognition model using BiLSTM-CRF network: https://blog.dominodatalab.com/named-entity-recognition-ner-challenges-and-model

2. Text Classification with BERT in Pytorch: https://towardsdatascience.com/text-classification-with-bert-in-pytorch-887965e5820f

3. Named entity recognition with BERT in Pytorch: https://towardsdatascience.com/named-entity-recognition-with-bert-in-pytorch-a454405e0b6a

4. BiLSTM original paper: https://paperswithcode.com/method/bilstm

5. Flair official Github Repo: https://github.com/flairNLP/flair

6. What are Word Embeddings: https://machinelearningmastery.com/what-are-word-embeddings/

7. A comparison of LSTM and BERT in small dataset: https://arxiv.org/abs/2009.05451

8. CodeBLEU: https://arxiv.org/abs/2009.10297

9. Evaluation, BLEU: https://cloud.google.com/translate/automl/docs/evaluate

10. Recall: https://developers.google.com/machine-learning/glossary#recall

11. Precision: https://developers.google.com/machine-learning/glossary#precision

Libraries

- os: https://docs.python.org/3/library/os.html
- pandas: https://pandas.pydata.org/
- numpy: https://numpy.org/
- re: https://docs.python.org/3/library/re.html
- nltk: https://www.nltk.org/
- spacy: https://spacy.io/
- textacy: https://textacy.readthedocs.io/en/latest/
- pytorch: https://pytorch.org/
- Levenshtein distance: https://pypi.org/project/python-Levenshtein/

- Longest common substring: https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher
- pickle: https://docs.python.org/3/library/pickle.html
- collection: https://docs.python.org/3/library/collections.html
- warnings: https://docs.python.org/3/library/warnings.html

Frame Definitions

1. {Verb: ‘Enter/Input/Set’, Value: ? (value to be entered), Location: ? (location to enter the value)}
2. {Verb: ‘Click’, Value: ? (the object to be clicked on)}
3. {Verb: ‘Wait’, Value: ? (the object to wait for), Time: ? (the amount of time to wait)}
4. {Verb: ‘Get’, Value: ? (the object to extract)}
5. {Verb: ‘Check’, Value: ? (the correct value to be checked against)}