# R1: The software is *effectively\** capable of recognizing faces of people, whom the model is trained on before, in a set of photos.

## Specification

Given a set of photos, the software should recognize faces in them. The software should recognize any number of faces in a picture as long as they are detected and the model knows them. The program should differentiate between the faces that cannot be detected, for low quality for example, and those that are detectable but unknown to the model.

For effective recognition, some parallelism could be considered when recognizing more than a photo for better performance.

This requirement was tested through both functional testing and structural testing, then for some changes to improve performance, it has undergone regression testing.

### Functional Testing

#### Test suite

A collection of photos was collected for people that the software was previously trained on some of them. The specification-based testing considered the possible scenarios that could occur with the program as can be seen in table 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test case ID | Test case description | Input (photo name) | Expected Output | Output | Status |
| 0 | Person 1 – known to the model | Princess Diana | * Princess Diana | * Princess Diana | Passed |
| 1 | Person 2 – known to the model | Dr. Ahmed Zewail | * Dr. Ahmed Zewail | * Dr. Ahmed Zewail | Passed |
| 2 | Person 3 – known to the model | Sir Magdi Yacoub | * Sir Magdi Yacoub | * Sir Magdi Yacoub | Passed |
| 3 | Person 4 – known to the model | Mohamed Salah | * Mohamed Salah | * Mohamed Salah | Passed |
| 4 | Person 5 (or more in bg) – NOT known to the model | Me | Unknown(s) | * Unknwon * Unknwon * Unknwon | Passed |
| 5 | Persons 6 and 7 - both known to the model | Ricciardone and Dallal | * Ricciardone * Dallal | * Dallal * Dallal | Failed |
| 6 | Persons 8 and 9 - both known to the model | P. Diana and Sir Yacoub | * P. Diana * Sir Yacoub | * Unknown | Failed |
| 7 | Persons 10 and 11 - both NOT known to the model | Mbappe and CR7 | * Unknown * Unknown | * Unknown * Unknown | Passed |
| 8 | Persons 12 and 13 - one unknown and one known | Hazard and Salah | * Unknown * Salah | * Unknown Salah | Passed |
| 9 | Non photo input | .csv file | Exception raised | Error | Failed |
| 10 | Non permitted input | Directory | Exception raised | Error | Failed |
| 11 | No persons | My home university | - | - | Passed |

Table 1

## Results

It is worth mentioning that testing the accuracy of the model was not the concern of this testing. It was done before as part of the component-based testing to check which machine learning model will be used for the purpose of this software, and the result was to opt a deep learning model implemented by the face\_recognition library.

Tests 5 and 6 were found to fail due to uncontrollable factors, in this unit. To explain, test 5 misrecognized on of the persons, which suggests that the model was not trained enough of its photo, and that was checked and found that the model used only one picture for the person even worse with a low quality. Test 6 failed because it could neither recognize a person known, which is the same reason for test 5, nor detect the other. The issue here is either because the model used is not the best, which is a component-based testing decision, or the photos used for training are few or with low quality.

Tests 9 and 10 failed because of a missing logic that the directory of photos to be recognized may contain non-image or unpermitted files/directories.

### White-Box Testing

The control flow of this unit was straight forward with only one if/else condition at the end of it to decide whether the person known or not. Statements executed behaved as expected; however, the condition of the if statement was partly incorrect, and could cause exceptions in some cases.

if check.count(True)/len(check) >= 0.8

consider the case when the length of the check list is 0. This was found in the documentation that is possible in some special cases that were not covered in the specification-based testing. As such, it was modified to the following statement:

if len(check) and check.count(True) / len(check) >= 0.8

As mentioned, executing statements was straightforward and achieving a 100% coverage could be done using a single photo that contains at least one unknown person and one unknown person, and they should be clear enough to be detectable as human faces.

### Additional Tests

To make the recognition of faces in multiple photos “efficient,” parallelizing the recognition unit was considered. As such, I used the multiprocessing concept to recognize each photo in a separate process whenever possible. After parallelizing the module, it has undergone the same tests of functional testing, and it produced the same results. I believe this could be considered a regression testing for this part.

A system test was done on the GUI for the same photos, and they produced the same results as can be seen in the figures below.

Graphical user interface, application

Description automatically generatedGraphical user interface, application

Description automatically generatedA picture containing text, screenshot, player

Description automatically generated

Figures 1-3

# R2: - The software stores the metadata entered by the user

## Metadata Format

* Name(s): If more than a person is to be tagged, names are to be separated by a special characted (for example a ';') with the same order shown on the squares of detected faces
* Photographer
* Date (YYYY-MM-DD)
* Place
* Event
* Description
* Source.

## Constraints and ITFs

There are only constraints on the *Date* and Name(s) entries. They are two *independently testable features*, so the specification could be decomposed for each of them. Other scenarios might have included more ITFs than those two, such as preventing empty entries, but for this software, this is allowed.

# 1. Date

## Specification

The date should be saved in ISO format as YYYY-MM-DD where year cannot exceed 2100 and cannot be below 1900. Set maximum days according to months and assume February 29 days. Program must tolerate any whitespaces.

The following test suite in table 2 can be derived to test the mentioned specification. As explained in the test cases descriptions, it focuses boundary values, which are especially trouble-prone regions in the code. The unittest module in Python was used to perform this functional testing.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test case ID | Test case description | Input | Expected Output | Output | Status |
| 0 | Check lower boundary of date (invalid) | 1889-12-31 | False | False | Passed |
| 1 | Check lower boundary of date (valid) | 1990-01-01 | True | False | Failed |
| 2 | Check upper boundary of date (valid) | 2100-12-31 | True | True | Passed |
| 3 | Check upper boundary of date (invalid) | 2101-01-01 | False | False | Passed |
|  |  |  |  |  |  |
| 4 | Check upper boundary in a 30-day month (invalid) | 2022-06-31 | False | True | Failed |
| 5 | Check upper boundary in a 30-day month (valid) | 2022-06-30 | True | True | Passed |
| 6 | Check upper boundary in a 31-day month (valid) | 2022-05-32 | False | False | Passed |
| 7 | Check upper boundary in a 30-day month (valid) | 2022-05-31 | True | True | Passed |
| 8 | Check lower boundary in a 31-day month (invalid) | 2022-05-00 | False | False | Passed |
| 9 | Check lower boundary in a 31-day month (valid) | 2022-05-01 | True | True | Passed |
| 10 | Check lower boundary in a 30-day month (invalid) | 2022-06-00 | False | False | Passed |
| 11 | Check lower boundary in a 30-day month (valid) | 2022-06-01 | True | True | Passed |
|  |  |  |  |  |  |
| 12 | Check upper boundary of months (invalid) | 2022-13-07 | False | False | Passed |
| 13 | Check upper boundary of months (valid) | 2022-12-07 | True | True | Passed |
| 14 | Check lower boundary of months (invalid) | 2022-00-07 | False | False | Passed |
| 15 | Check lower boundary of months (valid) | 2022-01-07 | True | False | Failed |
|  |  |  |  |  |  |
| 16 | Check different format (YYYY-DD-MM) | 2022-25-03 | False | False | Passed |
| 17 | Check different format (DD-MM-YYY) | 25-03-2022 | False | False | Passed |
| 18 | Check non-numerical values | YYYY-MM-DD | False | False | Passed |
| 19 | Check negative values - year | -199-11-11 | False | False | Passed |
| 20 | Check negative values – month | 2022--11-05 | False | False | Passed |
| 21 | Check negative values - day | 2022-11--5 | False | False | Passed |
| 22 | Check upper boundary of February (valid) | 2022-02-29 | True | True | Passed |
| 23 | Check upper boundary of February (invalid) | 2022-02-30 | False | True | Failed |
| 24 | Check whitespaces | 2 0 2 2 – 0 7 – 1 1 | True | True | Passed |
| 25 | Valid date (my DoB :”D) | 2000-11-18 | True | True | Passed |

Table 2

## Results

The status column was obtained from the results of unittest library in Python. As shown in figure 4, four failures were found. Tests failed are shown as 5 because the cases are considered sub-tests, and since at least one of them failed, then the main test failed as well, thus this one additional fail can be neglected.

Graphical user interface, text

Description automatically generated

Figure 4

Test cases 1 and 15 failed due to a logical operator in the following statement:

**if month > 12 or month <= 1:**

This statement validates that the month cannot be more than 12 or lower than 1, but as can be easily spotted, it should be “… or month < 1” where the equal sign caused this error to occur.

Test cases 4 and 23 failed because of a ***missing logic***. I forgot to handle the logic of number of days per month as 31, 30, or as assumed 29 days. This is a major advantage of black-box testing that discovers those missing parts that did not cover the specification.

## Fixing and re-testing

The logical operator was corrected, and the missing logic was added, then I re-tested the unit. As shown in figure 5, all tests passed after those modifications.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 5

# 1. Name(s)

## Specification:

A name can be of any format, and if there are multiple persons in the photo, names are to be separated by a semicolon ‘;’ with the same order shown on the squares of detected faces. That way, you can assume the user will not mistakenly include a ‘;’ within a name. If you want to skip one person, you can write the semicolon directly, and whitespaces are to be tolerated in this case. For example, person1; ; person3 shows that there are three people in the picture, and the first and third are the only ones to be tagged. You can end the line with a semicolon or not, but it is necessary when you want to skip the last person. In that case, person1; ; means two persons where only the first one is tagged. You cannot skip all persons in the photo.

The following test suite in table 3 is derived from the specifications. Although some of the test cases were redundant and checked on the same part of the specification, it was better to keep them separated to easily spot an error if there were any.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test case ID | Test case description | Input | Expected Output | Output | Status |
| 0 | No semicolon | Person1 | Person1 | Person1 | Passed |
| 1 | Semicolon | Person1; | Person1 | Person1 | Passed |
| 2 | Unknown person | Person1;; | Person1  ‘’ | Person1  ‘’ | Passed |
| 3 | Unknown person with spaces | Person1; ; | Person1  ‘’ | Person1  ‘’ | Passed |
| 4 | Unknown person with two other tagged persons | Person1;; person3; | Person1  ‘’  Person3 | Person1  ‘’  Person3 | Passed |
| 5 | Twp unknown persons with two tagged persons | Person1; ; person3;; | Person1  ‘’  Person3  ‘’ | Person1  ‘’  Person3  ‘’ | Passed |
| 6 | Starting with unknown and ending without a semicolon | ;; person3;;;;person7 | ‘’  ‘’  Person3  ‘’  ‘’  ‘’  Person7 | ‘’  ‘’  Person3  ‘’  ‘’  ‘’  Person7 | Passed |
| 7 | Multiple unknowns and a tagged person at the middle. Starting and ending with unknowns | ;;;;person5;; | ‘’  ‘’  ‘’  ‘’  Person5  ‘’ | ‘’  ‘’  ‘’  ‘’  Person5  ‘’ | Passed |
| 8 | Three unknowns. | ; ; ; | None | None | Passed |
| 9 | (repeated test) | ;person2 | ‘’  Person2 | ‘’  Person2 | Passed |
| 10 | Trailing whitespaces | ; ; person3 ; | ‘’  ‘’  Person3 | ‘’  ‘’  Person3  ‘’ | Failed |
| 11 | (repeated test) to attempt many unknowns versus only one known ending with a semicolon | ;;;;;;;;;person10; | ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  Person10 | ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  ‘’  Person10 | Passed |

Table 3

## Results

Test case 10 failed because of the trailing whitespaces. The logic in the implementation assumed that if there is a ‘;’ at the end, it would be the very last character. That was not true in that test case which caused this failure.

Fixing and re-testing

Once this issue is handled by removing the trailing spaces, this test case, along with the other tests was tested again and passed as shown in figure 6. This brings us to the concept of *regression testing* to ensure that the test cases still pass after any changes of the code.

A screenshot of a computer

Description automatically generated

Figure 6