



Handwriting²Text

CA400 Testing Manual

Project Title: Handwriting to Text Converter

Student Name: Conor Hanlon

Student Number: 15445378

Supervisor: Ray Walshe

Completed On: 18/05/19

Abstract:

Handwritten notes and essays are a key part of the academic environment. While the digital world is constantly growing, a lot of students, lecturers and professors still prefer writing notes by hand. My project is centered around converting a user's handwriting to a digital format.

I plan to create and train a neural network model that will be able to identify different characters and words in images uploaded by the user and convert this into a text document. The machine learning process will use convolutional and recurrent neural networks to process the training images. The model will be deployed and utilised by a simple web application. Any person can open up the application and upload images of their handwritten notes. They will be given the option to just download the generated document, or they can save it to their Google Drive if they are signed into their Google account. It will also be possible to share the document via email. This application will help users keep track of any handwritten notes or essays they may want to keep safe for future use.

Table of Contents

Unit Testing	3
Web Application	3
Flask Server	4
Integration Testing	7
Web Application	7
Flask Server	10
Acceptance Testing	10
Continuous Integration	12
Test Coverage	13
Model Validation	14
Heuristic Testing	14
Schneiderman's Eight Golden Rules	14
Strive for Consistency	14
Enable Frequent Users to Use Shortcuts	15
Offer Informative Feedback	15
Design Dialogue to Yield Closure	15
Offer Simple Error Handling	15
Permit Easy Reversal of Actions	16
Support Internal Locus of Control	16
Reduce Short-Term Memory Load	16
Nielsen's Heuristics	16
Visibility of System Status	16
Match Between System and the Real World	17
User Control and Freedom	17
Consistency and Standards	17
Error Prevention	17
Recognition rather than Recall	17
Aesthetic and Minimalist Design	18
Help Users Recognise, Diagnose and Recover from Errors	18
Help and Documentation	18
Accessibility Testing	19
Hearing Difficulties	19
Motor Skills	19
Colour Contrast Checks	19
Use Case Testing	21
User Testing	22

1. Unit Testing

Unit testing is where individual components in a code base are tested. The purpose is to validate that each unit of the software performs as designed.

For my unit testing, I used the Javascript framework QUnit to test my Ember.js code, and the Python module unittest for the Flask server.

Web Application

The vast majority of unit testing in the Ember.js framework was targeted at the home controller. The home screen directed most of the application's functionality, from loading in user documents already created to displaying components to logging out.

The functionality of displaying components had to be robustly tested to ensure multiple components were not being displayed simultaneously, which could cause issues with the application. I first started by simply testing the functions that displayed a single component without affecting others.

```
test('toggle loading', function(assert) {
  let controller = this.owner.lookup('controller:home');
  assert.ok(controller);

  controller.send('toggleLoading');
  assert.equal(controller.get('loading'), true);

  controller.send('toggleLoading');
  assert.equal(controller.get('loading'), false);
});
```

This function displays the loading screen overlay on the page when waiting for a response from the Flask API. It is independent of all other features, making it simple to test.

Next, I tested that functions passing parameters to components would work correctly as intended.

```
test('show results modal', function(assert) {
  let controller = this.owner.lookup('controller:home');
  assert.ok(controller);

  controller.send('displayResult', 'Test result.');
```

```

controller.send('hideResult');
assert.equal(controller.get('showResult'), false);
assert.equal(controller.get('result'), null);
});

```

This test case is targeted at the results modal, which displays the converted text to the end user. The test passes, showing that the result is successfully passed to the component.

I also had to test functions that toggle two components at once, as well as passing parameters. This is the case when transitioning between the document select component and the document sharing modal.

```

test('show share document modal', function(assert) {
  let controller = this.owner.lookup('controller:home');
  assert.ok(controller);

  controller.send('displayShare');
  controller.send('displayShareDocument', '12345678', 'Test Document');
  assert.equal(controller.get('showShare'), false);
  assert.equal(controller.get('showShareDocument'), true);
  assert.equal(controller.get('documentId'), '12345678');
  assert.equal(controller.get('documentTitle'), 'Test Document');

  controller.send('hideShareDocument');
  assert.equal(controller.get('showShareDocument'), false);
  assert.equal(controller.get('documentId'), null);
  assert.equal(controller.get('documentTitle'), null);
});

```

At the start of the test case, I call the *displayShare* function to set the relevant variables to their correct value. Calling the *displayShareDocument* function which should show the user the document sharing modal, also hides the document select modal, by setting *showShare* to false.

These tests ensure that the user can navigate throughout the application with ease and without issues.

Flask Server

The main aim of the Flask unit tests are to check the interactions with the MySQL database. I tested functionality like creating a user entry and checking that the user entry exists.

This involved creating a mock database for use during my testing. I configured SQLAlchemy to use memory to create the testing database, and then populated it with mock data for my test cases.

```
class DBTests(TestCase):

    def create_app(self):
        self.app = Flask(__name__)
        self.app.config['SQLALCHEMY_DATABASE_URI'] =
'sqlite:///memory:'
        self.app.config['TESTING'] = True
        self.app.register_blueprint(app_blueprint)
        return self.app

    def setUp(self):
        self.client = self.app.test_client()
        with self.app.app_context():
            db.init_app(self.app)
            db.create_all()
            self.populate()

    def tearDown(self):
        self.app = Flask(__name__)
        with self.app.app_context():
            db.init_app(self.app)
            db.drop_all()

    def populate(self):
        user1 = User(
            id='1',
            forename='Conor',
            surname='Hanlon',
            email='conorbh97@gmail.com',
            image_url='http://mock.url',
            access_token='MockToken1'
        )
        user2 = User(
            id='2',
```

```

        forename='John',
        surname='Smith',
        email='johnsmith@gmail.com',
        image_url='http://mock2.url',
        access_token='MockToken2'
    )
    doc1 = CreatedDocument(
        id='25',
        uid='1',
        title='Test Doc 1'
    )
    doc2 = CreatedDocument(
        id='26',
        uid='1',
        title='Test Doc 2'
    )
    db.session.add(user1)
    db.session.add(user2)
    db.session.add(doc1)
    db.session.add(doc2)
    db.session.commit()

```

From here, I could now proceed to carry out unit tests on my Flask server. The first test was simply to create a new user entry.

```

def test_create_new_user(self):
    data = {
        'id': '3',
        'forename': 'Tony',
        'surname': 'Stark',
        'email': 'stark@gmail.com',
        'imageUrl': 'http://mock3.url',
        'accessToken': 'MockToken3'
    }
    response = self.client.post('/create_user', data=data).data
    response_data = json.loads(response)
    assert response_data['val'] == 'New user created.'

```

This test hits the endpoint with the JSON data mimicking an API call from the web application. It carries out the functionality as normal and returns the response it would send as a result. The test passes, which confirms that the user entry was created in the mock database.

Next, I validated that the endpoint would check if the user logging in already exists in the database, as well as checking that it would update the access token appropriately. The tests are similar to the one above, ensuring that the correct response is the result of the API call.

```
def test_user_exists(self):
    data = {
        'id': '2',
        'forename': 'John',
        'surname': 'Smith',
        'email': 'johnsmith@gmail.com',
        'imageUrl': 'http://mock2.url',
        'accessToken': 'MockToken2'
    }
    response = self.client.post('/create_user', data=data).data
    response_data = json.loads(response)
    assert response_data['val'] == 'Access token up to date.'
```

```
def test_update_access_token(self):
    data = {
        'id': '2',
        'forename': 'John',
        'surname': 'Smith',
        'email': 'johnsmith@gmail.com',
        'imageUrl': 'http://mock2.url',
        'accessToken': 'MockToken22'
    }
    response = self.client.post('/create_user', data=data).data
    response_data = json.loads(response)
    assert response_data['val'] == 'Access token updated.'
```

2. Integration Testing

Integration testing is a level of software testing where components are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction and ensure all units can work together as intended.

Web Application

Due to the number of components I created in the web application, a lot of integration testing was necessary to cover all of the functionality. Some of these tests were straightforward, just validating that the correct text was being displayed on the user interface based on the component parameters.

```
test('it renders', async function(assert) {
  await render(hbs`{{page-header}}`);
  assert.equal(this.element.textContent.trim(), 'Login with Google');
  await authenticateSession();
  assert.equal(this.element.textContent.trim(), 'Logout');
});
```

Testing the page header component simply required checking that the button displayed changed between authenticating the session and logging out.

Due to my application using cookies to cache user information used by the web application, I had to figure out a method of mocking this during my testing. I used the stub approach, which allowed me to create a mock service. This would then return the mock data when called by the Ember components.

```
const cookieStub = Service.extend({
  currentUser:
  'Conor,Hanlon,conorbh97@gmail.com,1234,http://mock.url,mocktoken',
  clear () {
    return this.get('currentUser', null);
  }
});
```

One of the key goals of my integration testing was to make sure different buttons on the application carried out their intended functionality. This is apparent in my test case for the tutorial component, which has buttons to move between slides and also to a different tutorial.

```
test('it renders', async function(assert) {
  await render(hbs`{{tutorial-modal}}`);
  assert.equal($('.instructions ul li:first-child').text().trim(),
    'First, click the blue box indicated in the image above.');
```

```
  await click('.next-slide');
  assert.equal($('.instructions ul li:first-child').text().trim(),
    'You can upload more images to be converted by clicking the box again.');
```

```
  await click('.next-slide');
  assert.equal($('.instructions ul li:first-child').text().trim(),
    'The resulting text is displayed in the large box. This can be edited before saving the document.');
```



```

    await click('#labelB2');
    assert.equal($('.instructions ul li:first-child').text().trim(),
      'The application keeps track of each document you save to Google
      Drive, allowing you to share them\n' +
      '
      with your friends via email.');
```

```

    await click('.next-slide');
    assert.equal($('.instructions ul li:first-child').text().trim(),
      'You can add multiple email addresses to the input field shown
      above.');
```

```

    await click('.prev-slide');
    assert.equal($('.instructions ul li:first-child').text().trim(),
      'The application keeps track of each document you save to Google
      Drive, allowing you to share them\n' +
      '
      with your friends via email.');
```

```

    await click('#labelB1');
    assert.equal($('.instructions ul li:first-child').text().trim(),
      'First, click the blue box indicated in the image above.');
```

```

    await click('.close-button');
  });
});
```

Using QUnit's inbuilt `await` and `click` functions, I could ensure that the correct text was loaded into the user interface when the user makes a navigation action.

One of the more complex integration tests was verifying the image upload functionality. To carry out this test, I had to create a mock image file that would be accepted by the web application.

```

assert.equal($('h4').text().trim(), 'Upload');
let inputElement = $('.imageInput');

let blob = new Blob(['foo', 'bar'], {type: 'image/png'});
blob.name = 'foobar.png';
inputElement.triggerHandler({
  type: 'change',
  target: {
    files: {
      0: blob,
      length: 1,
      item() {
```

```

        return blob;
    }
}
});
await click('.imageInput');
assert.equal($('h4').text().trim(), '1 image(s) added.');
```

I created a custom Blob object that mimicked the user's image. As you can see in the test case above, the heading text changes to reflect the amount of images that were uploaded by the user. All of these integration tests were crucial to ensuring the system is reliable as possible with the core functionality.

Flask Server

Integration tests were also necessary in the Flask server. The tests involved mocking the Google Drive API for the relevant endpoints. This required a function that would return an example document ID from the API.

```

def mock_google_create(a, b, c):
    return "87654321"

@patch('app.create_doc', side_effect=mock_google_create)
def test_create(self, mock_create):
    data = {
        'filename': 'TestFile5',
        'content': 'A document for my integration tests.',
        'accessToken': 'MockToken1'
    }
    response = self.client.post('/create', data=data).data
    response_data = json.loads(response)
    assert response_data['val'] == 'Google Doc Created'
```

Other similar tests were created for the rest of the relevant endpoints. These tests ensured that the server handles the Google API responses appropriately, by creating a database entry and sending the correct response the web application.

3. Acceptance Testing

The purpose of acceptance testing is to evaluate the system's compliance with the initial requirements and assess whether it is providing the necessary features at a high standard.

To ensure that the acceptance tests as well as the integration tests would run similarly to the application when deployed, I used Mirage.js to mock the API responses.

```
this.urlPrefix = 'http://localhost:5000';

this.get('/documents/:user_id', {
  val: [
    { 'id': 'mock_id1', 'uid': 'mock_uid1', 'title': 'TestDB8',
      'created_on': '2019-03-21 19:05:41' },
    { 'id': 'mock_id2', 'uid': 'mock_uid2', 'title': 'TestDB7',
      'created_on': '2019-03-21 13:13:25' }
  ]
});

this.post('/create_user', {status: 'User Created'});

this.post('/create', {status: 'Document created successfully.'});

this.post('/share_document', {status: 'Document shared successfully.'});

this.post('/upload', {val: 'Sample result text.'});
```

Mocking these response allowed me to test the application in a way that would produce as realistic of a scenario as possible. To ensure that the code hits the Mirage endpoints instead of making an actual call, I configured the acceptance tests to use the Mirage hooks.

```
module('Acceptance | callback', function(hooks) {
  setupApplicationTest(hooks);
  setupMirage(hooks);

  hooks.beforeEach(function() {
    this.owner.register('service:cookies', cookieStub);
  });

  test('visiting /callback', async function(assert) {
    await authenticateSession();
    await visit('/callback#token=mock_token&type=mock_type');

    assert.notEqual(currentURL(),
      '/callback#token=mock_token&type=mock_type');
```

































```
});  
});
```

This test verifies that the application carries out functionality on the Google login callback, and then proceeds to the home page. We see that because the route has changed, which is the result of a mocked API call to the Flask server. This and other acceptance tests ensure that the initial specifications of the design are met and the application flows as intended.

4. Continuous Integration

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository in smaller units frequently. Each check-in is then verified by an automated build, allowing teams to detect problems early.

I made use of Gitlab CI during the development stage of my final year project. With each commit, a Gitlab runner would execute the commands and create the environment in my `.gitlab-ci.yml` file. This would run a project build and execute my tests.

11 Apr, 2019 2 commits			
	Add Flask integration/unit tests Conor Hanlon authored 1 month ago		c2ff1d38  
	Upload improved model (86% accuracy) Conor Hanlon authored 1 month ago		d267b47f  
09 Apr, 2019 3 commits			
	Add Blog 11 Conor Hanlon authored 1 month ago		5b46f37e  
	Add Frontend acceptance/integration tests Conor Hanlon authored 1 month ago		a133936f  
	Add Frontend acceptance tests Conor Hanlon authored 1 month ago		884f68a1  
08 Apr, 2019 1 commit			
	Add Flask integration tests Conor Hanlon authored 1 month ago		61f5fd54  
03 Apr, 2019 1 commit			
	Refine image processing Conor Hanlon authored 1 month ago		9661aced  
02 Apr, 2019 2 commits			
	Remove segmented words after prediction Conor Hanlon authored 1 month ago		c6da5b73  

Each build would either pass or fail, indicated by the green and red circles alongside the commit. Gitlab allows you to click into the build logs to find out the reason for the failure.

```

ok 6 Chrome 73.0 - [168 ms] - Integration | Component | upload-modal: it renders
ok 7 Chrome 73.0 - [54 ms] - Unit | Controller | application: it exists
ok 8 Chrome 73.0 - [43 ms] - Unit | Controller | home: it exists
ok 9 Chrome 73.0 - [47 ms] - Unit | Route | application: it exists
ok 10 Chrome 73.0 - [54 ms] - Unit | Route | callback: it exists
ok 11 Chrome 73.0 - [43 ms] - Unit | Route | home: it exists
ok 12 Chrome 73.0 - [46 ms] - Unit | Route | login: it exists
ok 13 Chrome 73.0 - [2 ms] - ember-qunit: Ember.onerror validation: Ember.onerror is functioning properly

1..13
# tests 13
# pass 12
# skip 0
# fail 1
Testem finished with non-zero exit code. Tests failed.

ERROR: Job failed: exit code 1

```

As indicated in the error message above, this particular build failed due to a test failure. Continuous integration is crucial to development, as continuously running tests help us ensure that new bugs don't manifest with our changes, just like in this scenario. This process helped me with quality control and removing faults from the application.

5. Test Coverage

Code coverage is an important metric in checking to what extent the functionality of the application has been validated. If large chunks of code are not being tested, there is no guarantee that they will work as designed in production.

To test my code coverage, I used an Ember.js module that reported statistics on how many lines of code I had tested as well as all the various branches I that had been followed. Below are the test statistics for the web application.

/									
90.58% Statements 125/138 76.47% Branches 26/34 90.32% Functions 28/31 90.58% Lines 125/138									
File	Statements	Branches	Functions	Lines					
app/	100%	7/7	100%	0/0	100%	1/1	100%	7/7	
app/adapters/	33.33%	1/3	100%	0/0	0%	0/1	33.33%	1/3	
app/authenticators/	100%	0/0	100%	0/0	100%	0/0	100%	0/0	
app/components/	85.71%	66/77	76.67%	23/30	85.71%	12/14	85.71%	66/77	
app/controllers/	100%	25/25	100%	2/2	100%	12/12	100%	25/25	
app/routes/	100%	26/26	50%	1/2	100%	3/3	100%	26/26	
app/templates/	100%	0/0	100%	0/0	100%	0/0	100%	0/0	
app/templates/components/	100%	0/0	100%	0/0	100%	0/0	100%	0/0	

The tests I have written cover 90% of the code, which is quite a comprehensive test set. From these results, I can move forward with the idea that the application will have an extent of stability when running in a production environment.

6. Model Validation

The neural network is the core component of my final year project. A thorough validation process is necessary to ensure it makes accurate predictions and is of a high standard. As mentioned in my technical manual, I am using the IAM handwriting dataset. This consists of 115,000 images of handwritten words.

After the training process, it was time to validate my model to check its accuracy. I split the dataset, using 75% for training and 25% for the validation process. The error was calculated by getting the distance between the network output and the ground truth text, and this value was used to calculate the precision of the output. My final model reached an accuracy of approximately 86%.

```
Batch: (17294 / 17298)
Batch: (17295 / 17298)
Batch: (17296 / 17298)
Batch: (17297 / 17298)
Batch: (17298 / 17298)
Batch: (17299 / 17298)
Character error: 5.573033707865169
Line accuracy: 86.0330674066366
```

This is quite a high accuracy, which begs the question of whether or not the model is overfitted. Overfitting is when a model works well for the training data, but fails to generalise new scenarios. It learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data.

To reduce the possibility of overfitting my model, I applied random stretches to my data during the training process. The stretched image was then fitted back into the 128x32 shape, keeping the input in the required format for the convolutional neural network. Constant exposure to “new” images helps the model learn more from the inputs rather than repeatedly passing the exact same picture.

7. Heuristic Testing

7.1. **Schneiderman’s Eight Golden Rules**

- *Strive for Consistency*

Buttons, menus and icons should remain consistent throughout the application. The web application uses common navigation symbols and a common colour scheme to ensure the next step is always intuitive for the user.

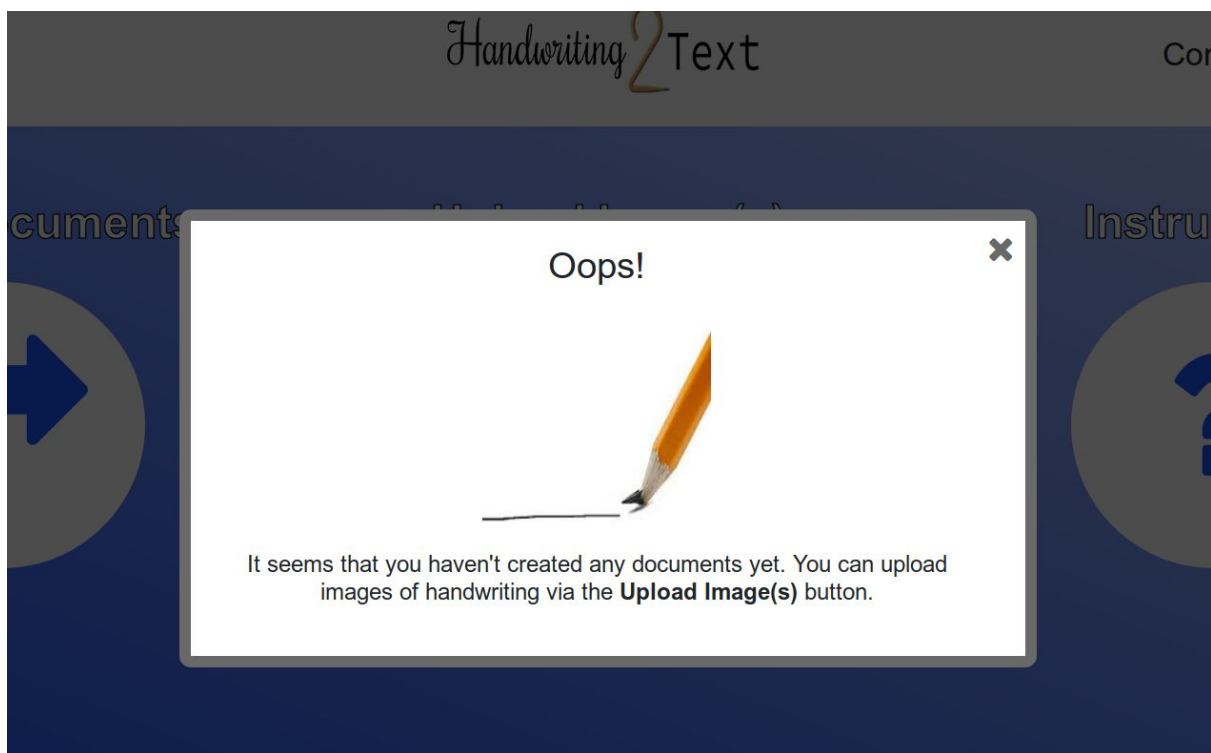


- *Enable Frequent Users to Use Shortcuts*

In general, users who frequently use the system should be able to take shortcuts where available. Unfortunately in the case of this application, the processes for sharing documents and uploading images is linear and all steps must be followed. Despite this, both processes are straightforward and only require a couple of steps.

- *Offer Informative Feedback*

Different dialogue boxes and buttons offer information about the current status of the system, which helps the user understand the situation. For example, they are told the reason why there are no documents available to share if there have been none created.



- *Design Dialogue to Yield Closure*

The dialogue in various buttons throughout the application give closure on what will happen if they are clicked. This way the users can make informed decisions on what they want to do.



- *Offer Simple Error Handling*

The web interface offers feedback to a user when they are trying to perform an invalid action. Various error messages are displayed outlining the issue.



Filename:

Please enter a filename above.

- *Permit Easy Reversal of Actions*

Different user actions can be reversed throughout the use of the application. The best example is being able to remove entered email addresses when sharing a document. Being able to reverse actions ensures the user isn't on edge about making mistakes that can undo a lot of effort.



× test.email@gmail.com

- *Support Internal Locus of Control*

The application offers users a degree of choice when it comes to what end goal they want to follow. They can choose to save a document locally or create a Google Doc on their Drive. As well as this, they have a choice of created documents to share, as well as the option to choose what document access to grant their friends.



Writer Commenter Reader

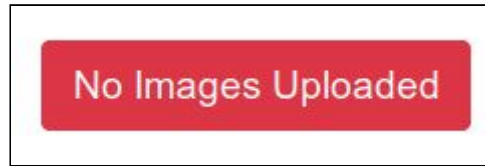
- *Reduce Short-Term Memory Load*

The interface itself is quite intuitive and easy to use. It does not require users to vigorously learn instructions beforehand to reach the end goal. If in any case they do get confused and are not sure what action to take, the tutorial modal provides details on how the application works step by step.

7.2. Nielsen's Heuristics

- *Visibility of System Status*

The user interface offers informative feedback about the current status. This ensures the user is never in the dark about what operations they should take or what processes are currently being carried out. The below button indicates the current state, showing that a user cannot proceed.



- ***Match Between System and the Real World***

The languages and phrases used by the application are clear, simple and easy to understand. The language of the user is apparent, rather than technical jargon which may be confusing.

- ***User Control and Freedom***

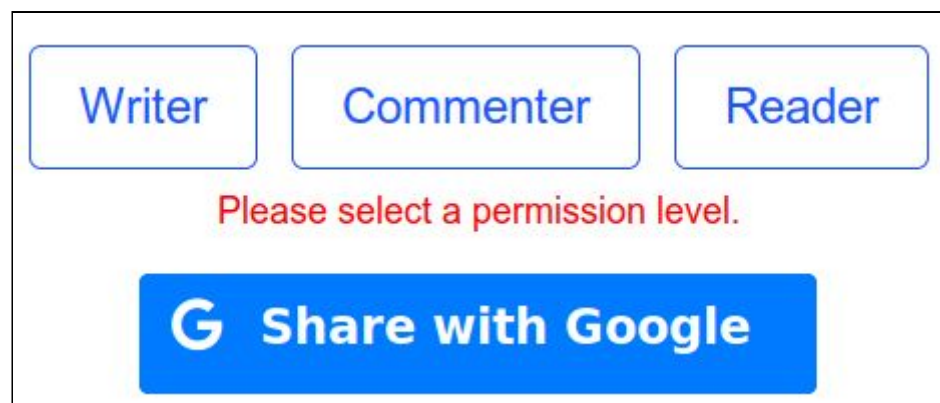
At any point during operations, a user can choose to close the current pop up modal and start again or perform a different action. They can freely browse the interface, without being forced to convert text or share documents.

- ***Consistency and Standards***

Icons and buttons use a consistent style throughout the interface. The colour scheme of blue and white is commonly used and is apparent in most components.

- ***Error Prevention***

As mentioned earlier in Schneiderman's Golden Rules, the web application prompts users with error messages when they try to perform an illegal action.








- ***Recognition rather than Recall***

Consistency throughout the application makes it easier for users to become familiar with different operations rather than learning exactly how every single feature works.

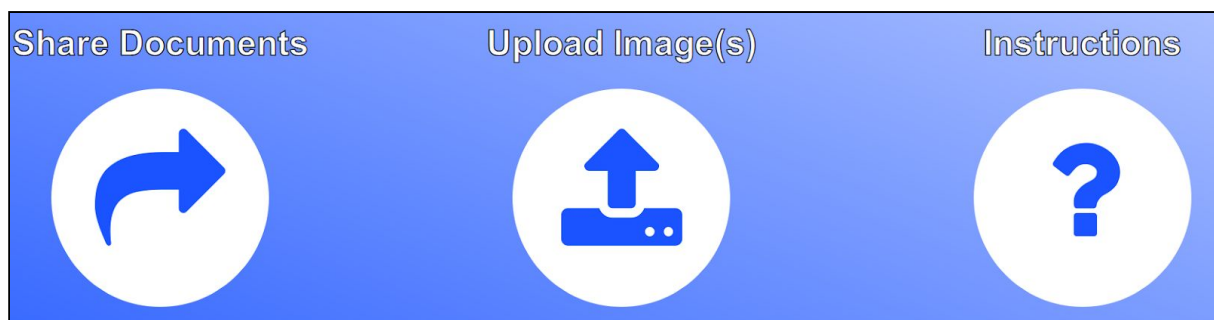
- ***Flexibility and Efficiency of Use***

As mentioned previously, the processes are short and linear so they can not be skipped by users. They can however, choose what operation to perform, such as saving a document to Google Drive versus downloading a local file. Over time, they will also build up a list of documents, which they have a choice of selecting to share.

2019-03-21 19:05:41	TestDB8	
2019-03-21 13:13:25	TestDB7	
2019-04-10 17:36:28	TestFile1	
2019-03-21 12:45:05	TestDB5	
2019-03-21 12:44:50	TestDB4	

- *Aesthetic and Minimalist Design*

Only necessary information should be provided to user. Too much content at once can be overwhelming and hinder progression through the application. The home screen has a simple design, that doesn't bombard the user with unnecessary information.

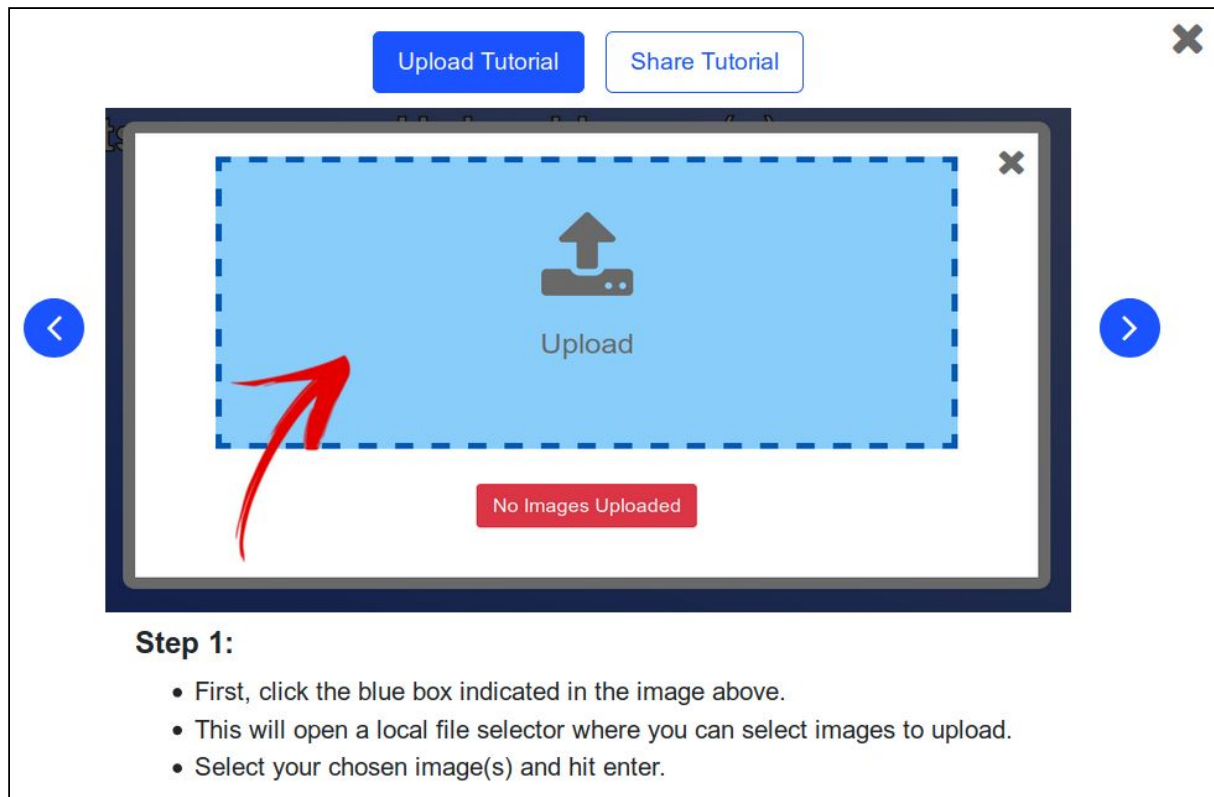


- *Help Users Recognise, Diagnose and Recover from Errors*

As mentioned previously, different error messages are displayed to the user to stop any operations that are not permitted.

- *Help and Documentation*

At any time when using the web application, the user may view the tutorial pop-up which walks them through the different operations step by step.



8. Accessibility Testing

8.1. Hearing Difficulties

As there no audio within the application their will be no difficulties present for users with hearing difficulties.

8.2. Motor Skills

Buttons throughout the application are large and easy for users to use. Items which are to be used in sequence are also placed close together which benefits these users.

8.3. Colour Contrast Checks

For all users and individuals who are visually impaired especially, it is crucial that the colour contrast on the web application does not inhibit their ability to use the system.

I used an online colour contrast checker to ensure that my design is accessible to as wide an audience as possible. The results were positive overall, with one small issue regarding the selected red background colour. From there I was able to adjust the value so that it passed the online check, furthering the accessibility of the application.

Color Contrast Checker

[Home](#) > [Resources](#) > Color Contrast Checker

Foreground Color

#FFFFFF

Lightness



Background Color

#C3201D

Lightness



Contrast Ratio

5.92:1

[permalink](#)

Normal Text

WCAG AA: **Pass**

WCAG AAA: **Fail**

The five boxing wizards jump quickly.

Large Text

WCAG AA: **Pass**

WCAG AAA: **Pass**

The five boxing wizards jump quickly.

Graphical Objects and User Interface Components

WCAG AA: **Pass**

Text Input

Color Contrast Checker

[Home](#) > [Resources](#) > Color Contrast Checker

Foreground Color

#0036D6

Lightness



Background Color

#FFFFFF

Lightness



Contrast Ratio

8.4:1

[permalink](#)

Normal Text

WCAG AA: **Pass**

WCAG AAA: **Pass**

The five boxing wizards jump quickly.

Large Text

WCAG AA: **Pass**

WCAG AAA: **Pass**

The five boxing wizards jump quickly.


Graphical Objects and User Interface Components


WCAG AA: **Pass**

Text Input

Color Contrast Checker

[Home](#) > [Resources](#) > Color Contrast Checker

Foreground Color
#FFFFFF
Lightness


Background Color
#0036D6
Lightness


Contrast Ratio
8.4:1
[permalink](#)

Normal Text

WCAG AA: **Pass**

WCAG AAA: **Pass**

The five boxing wizards jump quickly.

Large Text


WCAG AA: **Pass**

WCAG AAA: **Pass**

The five boxing wizards jump quickly.

Graphical Objects and User Interface Components

WCAG AA: **Pass**


Text Input

9. Use Case Testing

A use case is a description of a particular use of the system by an actor (a user of the system). Each use case describes the interactions the actor has with the system in order to achieve a specific task (or, at least, produce something of value to the user). I previously defined the use cases for the application in the functional specification.

To test these use cases and ensure the application works as intended, I stepped through the scenarios step by step, carrying out the described tasks. This would allow me to view the system from a user's perspective, and find any possible issues.

Reference Number	Scenario	Result	Developer's Comments
001	Login to Application	Success	
002	Logout of Application	Success	
003	Upload Image(s) of handwriting	Fault - Only converted one of three images	All images are being sent, take corrective action in Flask server
004	Save to Google Drive	Success	
005	Download Locally	Success	
006	Share via Gmail	Fault - Always granting edit access	Same value being sent from web application for all three options
007	View Tutorial Slides	Success	

Walking through the manual use cases helped me uncover numerous bugs which impacted some of the application's core functionality. Performing these steps as early as possible helped me eradicate these faults and ensure as a high a quality service as possible.

10. User Testing

User testing is the process through which the interface and functions of a website, app, product or service are tested by real users. It is one of the most important validation techniques, as it provides constructive criticism of the application in the areas that matter most to the target audience.

I carried out user testing with 14 different users. It varied from classmates to people I knew from different universities. It was mainly aimed at students because they are the primary target of the service. Before starting the testing, I provided them with all necessary information regarding the nature of the survey. They would not need to provide any private information except their email address, and I would share the results with each individual upon completion. The user survey involved is also completely anonymous. In compliance with GDPR, all results and data will be destroyed at the end of the project examination. I had filled out the necessary documentation and received approval from the School of Computing ethics board to move forward with the procedure, which I also disclosed with participants.

The first activity I carried out was use case testing. Each individual was given a set of instructions for the use cases mentioned above, as well as a passage of text to write out in their handwriting and upload to the system. Their objective was to follow the

steps and carry out the given tasks without any extra input from myself. I timed each participant as they completed the process, so that I could compare my expected time to the actual time required for each task. The top line of the below table references each use case based on the numbers on the previous page. All values are in the unit of seconds.

User Number	001	002	003	004	005	006	007
1	21	4	32	9	9	32	40
2	23	3	34	13	12	32	44
3	18	6	29	11	10	29	46
4	34	6	35	11	10	37	43
5	21	5	34	12	9	33	43
6	14	7	28	9	8	28	39
7	46	3	30	14	11	38	53
8	19	4	30	18	12	30	40
9	22	4	34	10	8	32	42
10	24	3	42	10	9	32	45
11	16	5	31	12	11	31	45
12	28	6	34	8	7	35	48
13	20	4	29	11	10	28	43
14	22	8	32	10	10	35	42
Expected Time	20	5	25	10	10	30	45

The results of these of tests gave me a great insight into how the intended users were able to learn the system and perform operations. For the most part, the results were in a range close to my predictions. There were a few outliers, for example users forgetting their email address password which slowed down the login process. My estimation was wrong for the uploading an image use case. It seemed users were slightly unfamiliar with the file explorer and this delayed adding pictures to the system. Overall, carrying out these tests helped me evaluate my use cases and ensure that they are feasible.

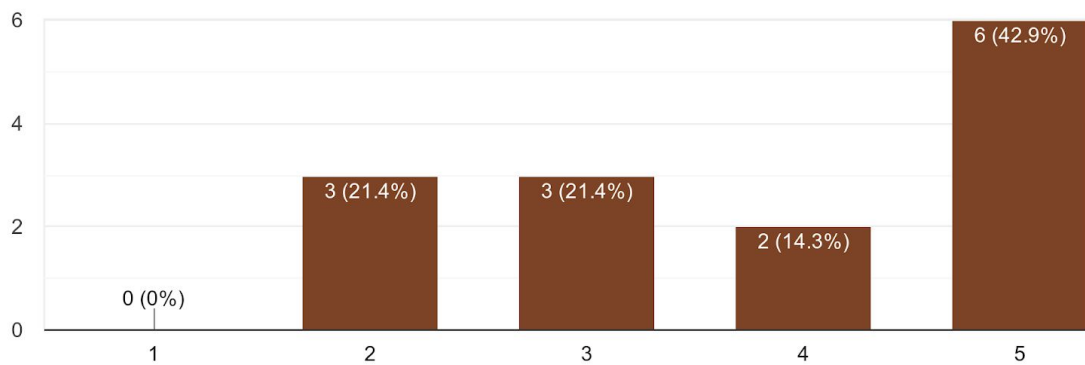
Once the use case testing had been completed, the participants were asked to fill out a short anonymous survey regarding their experience with the application. This allowed me to assess the system in a qualitative manner as well as the previous

quantitative manner. Both aspects are important in developing as appealing a system as possible.

The survey began by asking questions regarding the participants opinions regarding handwritten notes in general.

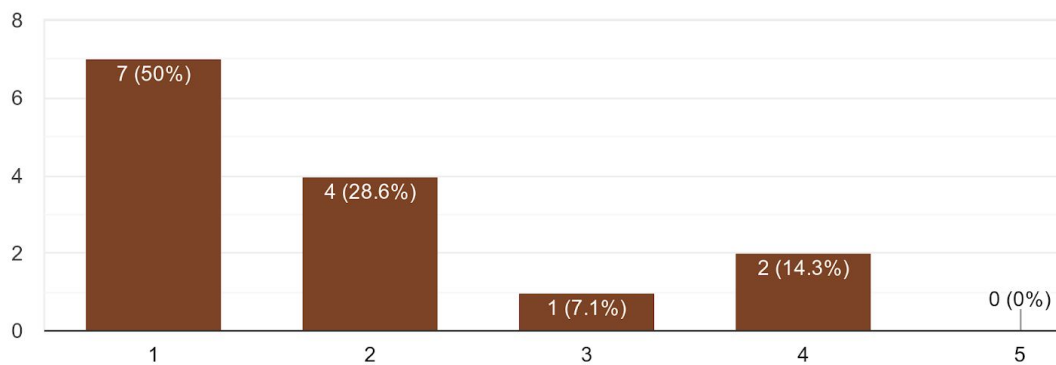
How often do you make handwritten notes?

14 responses



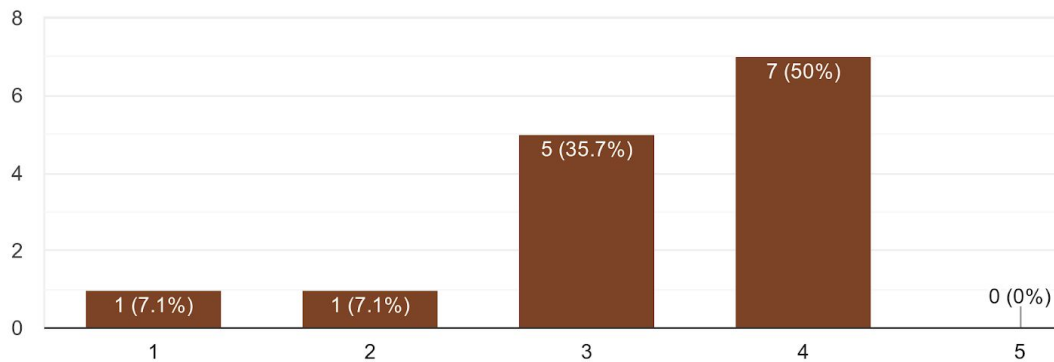
How often do you type up your handwritten notes?

14 responses



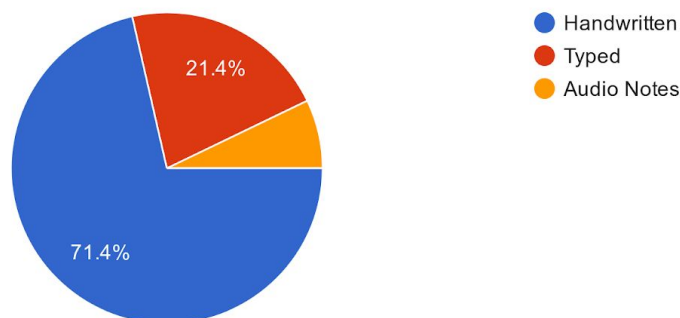
How often do you misplace handwritten notes?

14 responses



Do you prefer writing notes by hand or typing them?

14 responses

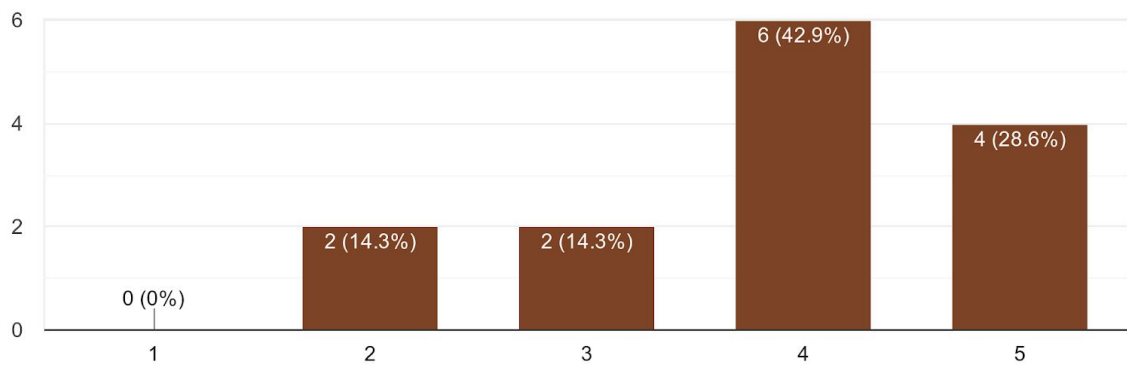


The general consensus from these questions was that the majority of users prefer writing notes by hand, and do so quite often. Some participants typed notes instead, and one volunteer even uses audio notes as their preferred study method. It is also apparent that it is common to misplace or lose notes, which may have important information regarding exams or course content. These questions helped me understand the viewpoint of the user, and also verified that this is an application that would be very useful to students.

Next, I asked different questions regarding the web application itself. It is important to receive this opinionated feedback, as it provides an understanding of what can be improved. I asked for suggestions on what features could be added, as well as what the participant's favourite features were.

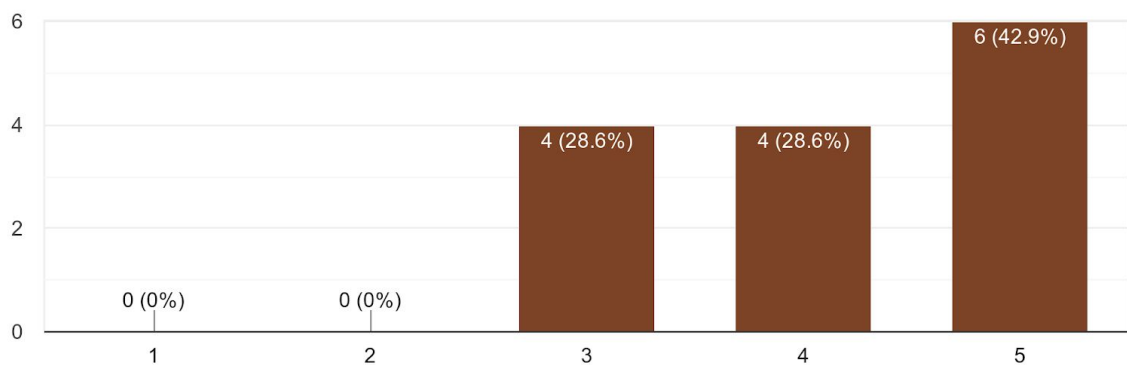
I would use this web application.

14 responses



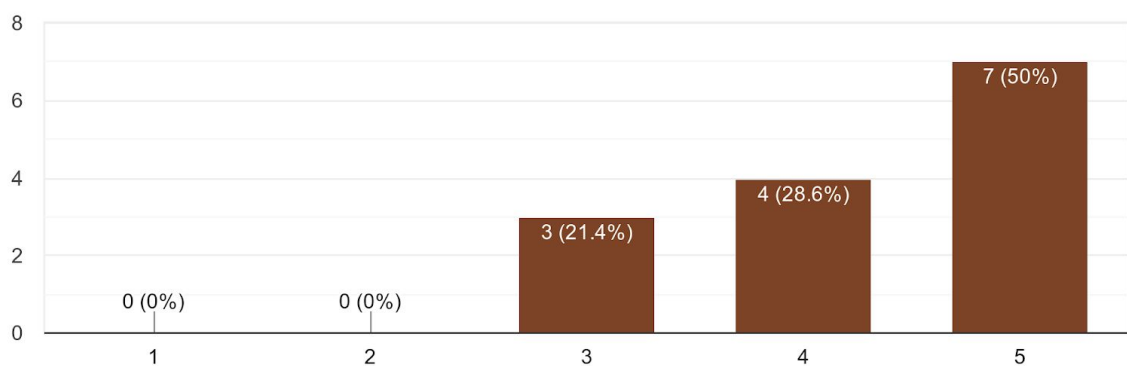
I would recommend this web application to a friend

14 responses



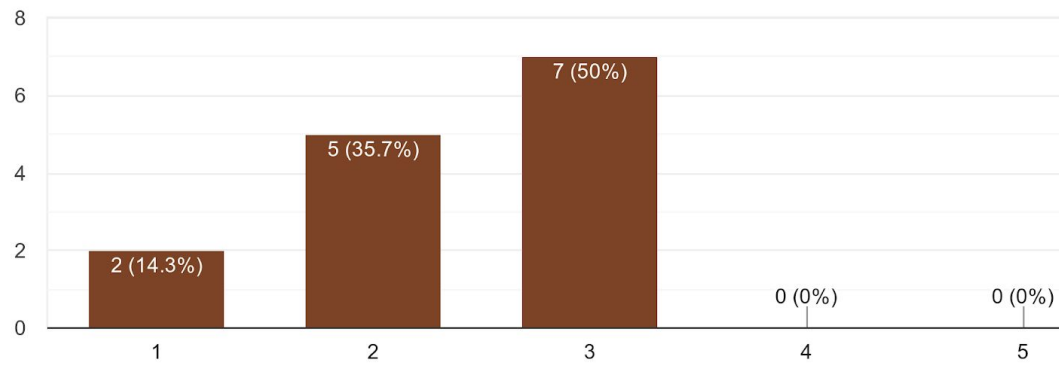
I like the design of the web application

14 responses



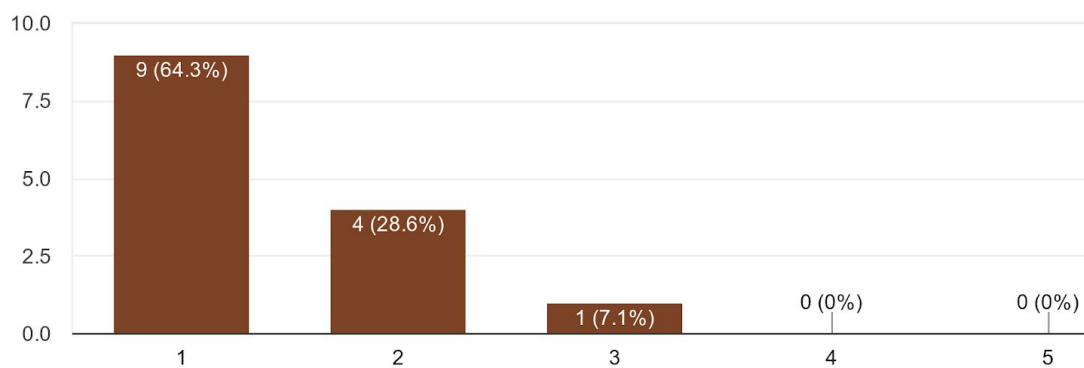
There are parts of the web application that could be improved

14 responses



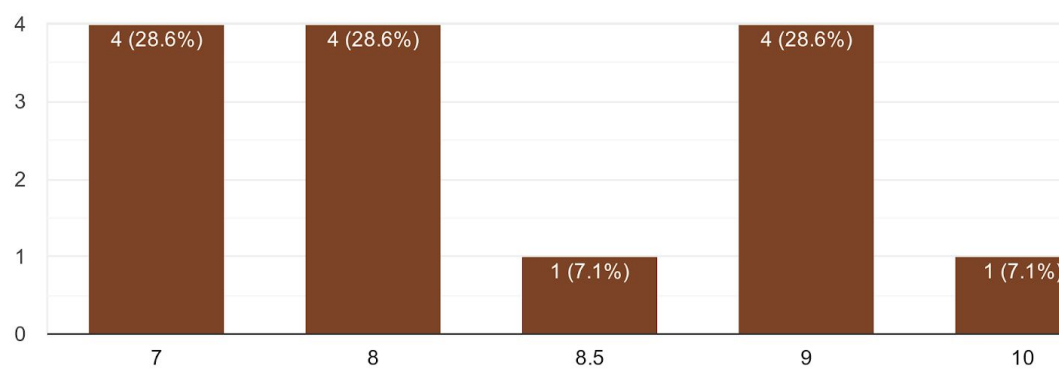
I found this web application difficult to use

14 responses



What score do you give the web application out of 10?

14 responses



From the results in the above charts, the results were positive overall. Users felt the application was helpful and easy to use. However, the majority of participants did feel that there was room for improvement. When I asked the question "*Is there anything you feel is missing from the web application?*", I had multiple different ideas put forward.

A common response was that there should be a feature that convert diagrams to a Google Drawing as well, as diagrams are common in student's notes. I had considered this option throughout the development process, but after some investigation I realised that this would just be infeasible in the time frame allocated for the project. Given a longer deadline in another scenario, I would definitely focus on adding features such as this one. Another common request was to identify headings and bullet points. Again, this was a feature I looked into but did not have the time resources available to implement.

Overall the user testing was very beneficial in getting into the mindset of a new user of the system. I got informative feedback about the feasibility of my defined use cases as well as eye-opening insights into user opinions regarding the interface itself. All of this primary data was crucial in allowing me to validate my system as a whole.