Technical Guide

CA400 Final Year Project

Project Title:	OCR Text to Speech Reading Aid
Student 1:	Róisín O'Rourke - 19360491
Student 2:	David Weir - 19433086
Supervisor:	Paul Clarke
Date Completed:	06/05/2023

Abstract

The project is an OCR based text to speech aid. It is designed to help users convert images of text into PDFs, or MP3 files that can be listened to using an MP3 player on the application. It can scan images of printed text or limited handwritten text, and can also accept PDFs. The application also has translation capabilities for French and German, for users who wish to translate their uploaded documents. It is aimed towards students as an aid to help them study by listening to their notes and converting them into PDFs.

The project is designed using python and has a GUI that was created using TKinter.

Table of Content

1. Motivation

The application we have created is an OCR reading aid. It has the capacity to scan printed and handwritten text, detect the text language, translate the text into a set sample of languages and summarise it and then finally convert the text into an mp3 or PDF format. The text to speech functionality will allow users to convert handwritten documents into pdfs and listen to them in audio format.

Our goal was to create an application that would be able to assist students in their studies by allowing them to convert their notes into new formats, be that an audio recording of them, or the notes converted into a PDF. As two students we understand the pressures of exams and the importance of good study practices. According to the CPD, benefits of audio learning include increased information retention and it provides the ability to multitask [1]. Similarly, studies conducted by Sheffield Hallam University and the University of Sheffield highlight the benefits of audio learning for international students [2]. Our application aims to aid students with their learning by providing them with helpful tools to convert notes into audio format, and also for international students to be able to convert notes into their native language and use them in their studies.

2. Research

Before beginning on the development of the application, several different avenues were explored on how to achieve the key functions of the project. We researched different approaches for the different components before choosing the ones we believed would help us best meet our functional requirements.

We decided on python due to its versatility and its extensive imaging processing and natural language processing capabilities.

Once we had decided on python as the programming language of the application, we researched which framework to use to create the GUI of the application. Several are available for python such as Tkinter, PyQt5 and Kivy. We decided to implement the GUI using Tkinter, as it supports a wide variety of widgets and is already installed in Python, and it does not require any sort of licensing, unlike PyQt5.

One of the key functionalities of the application is translation. We research what method to use to implement the translation for the app. Options included using Google

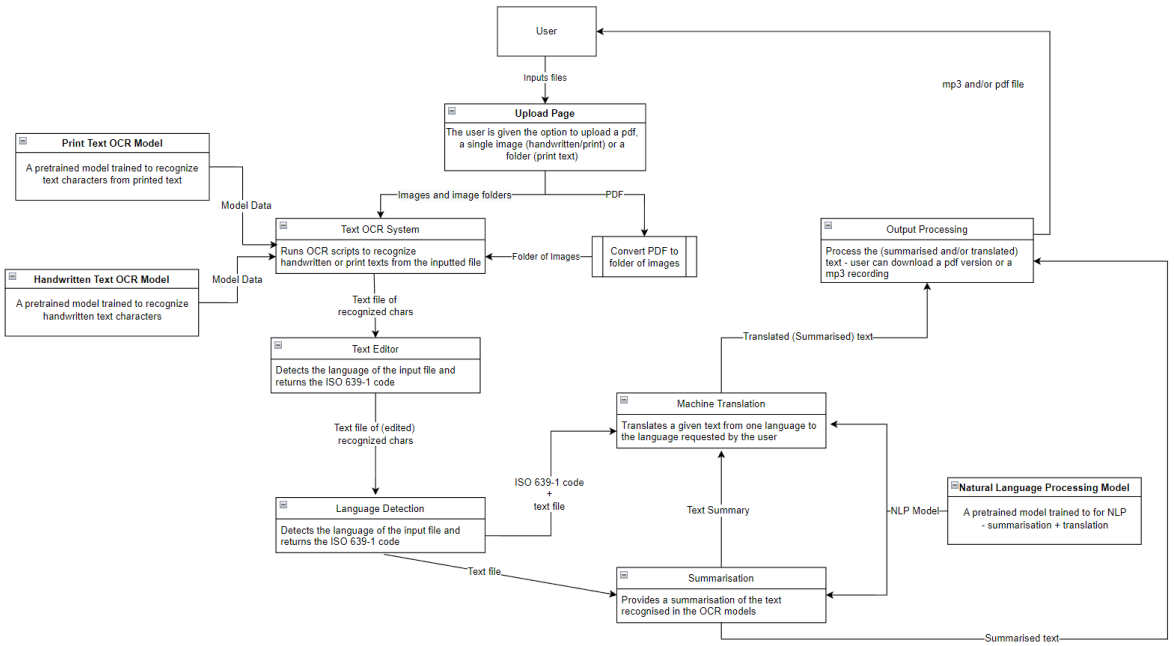
translate, huggingface transformers, or creating and training our own translation model. Each option has a varying degree of difficulty. While Google translate will generate a translation the quickest, its accuracy does not match what can be achieved using transformers or a self trained model. Ultimately, Hugging Face transformers were chosen as they represent a compromise, both in efficiency and complexity, between Google translate and a self trained model. Upon researching Hugging Face transformers more, we discovered the other potential uses it could have for our project, and implemented its summarisation capabilities alongside the translation.

An original intended purpose of the application was to serve as an aid to the visually impaired. However, after researching what additional features would need to be added to the application to achieve this purpose, we soon realised it lay outside the scope of the project.

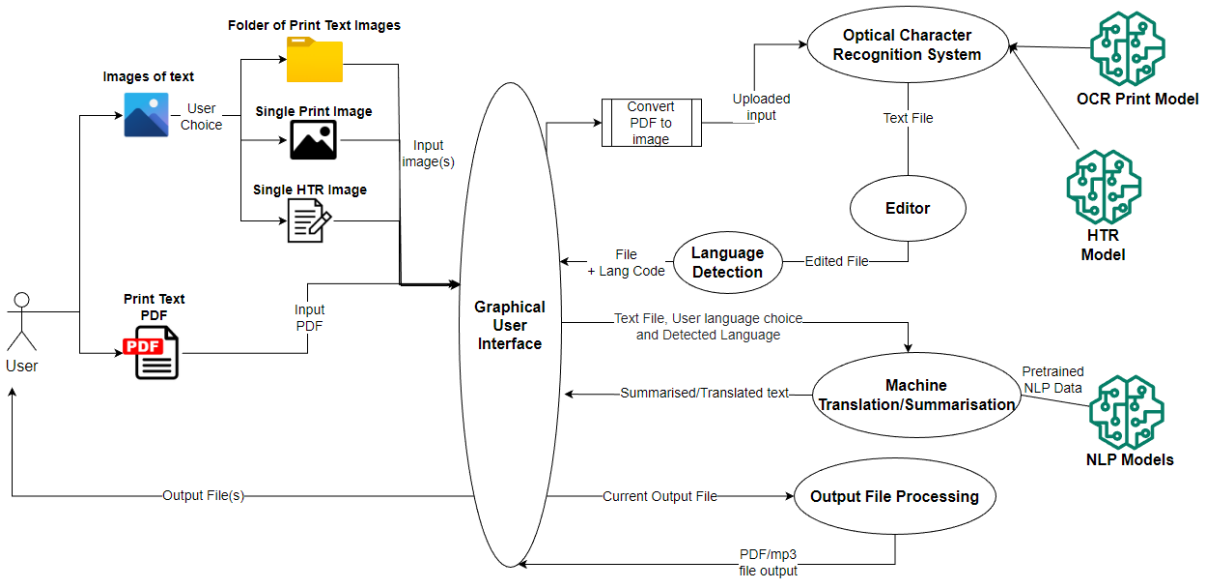
3. Design

3.1 High-Level Design

3.1.1 Data-Flow Diagram (DFD)



3.1.2 Context Diagram



3.2 Frontend Design

When planning the front end design, we wanted to follow design guidelines to create an application that is simple, straightforward and easy to use. The application has a layout, where the user moves from page to page and each page represents a step of the process. This allows users to move through the core functionalities in a logical manner with similar tasks being grouped together. The process of the application is as such:

Upload document(s) → Make any edits to the text → Modify the text → Download the text

The front end was designed to achieve key UI design guidelines [3]. The appearance and layout of the application is consistent throughout. Each page follows the same layout, with a heading, main central frame and a bottom frame with buttons to move between frames. The background and buttons of each frame are a consistent colour.

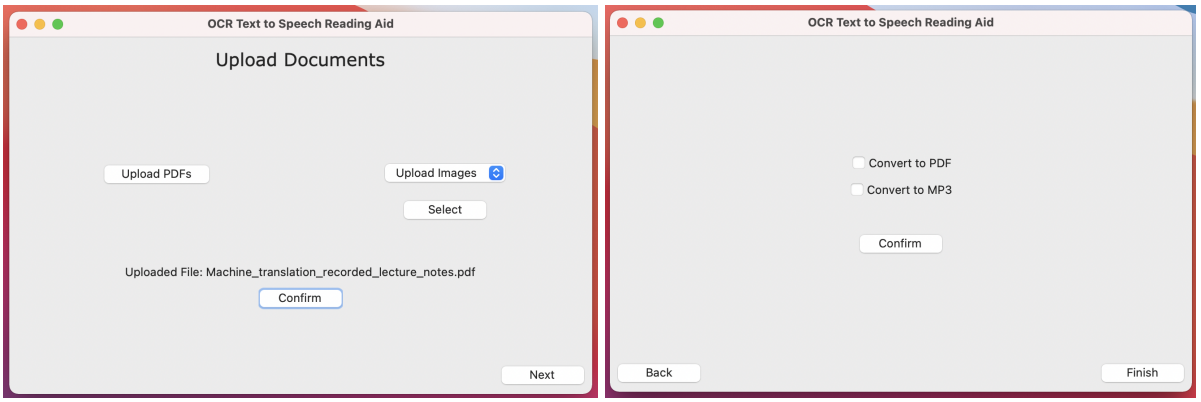


IMAGE ONE & TWO: Screenshots of the Application's GUI

The design of the user interface is simple and does not contain any unnecessary items or widgets that are not directly tied to the functionality of the application. This will minimise the cognitive load for the users and result in minimal confusion and a clear step by step process of moving through the application.

3.3 Backend Design

3.3.1 Sequence Diagrams

3.3.1.1 Upload Documents

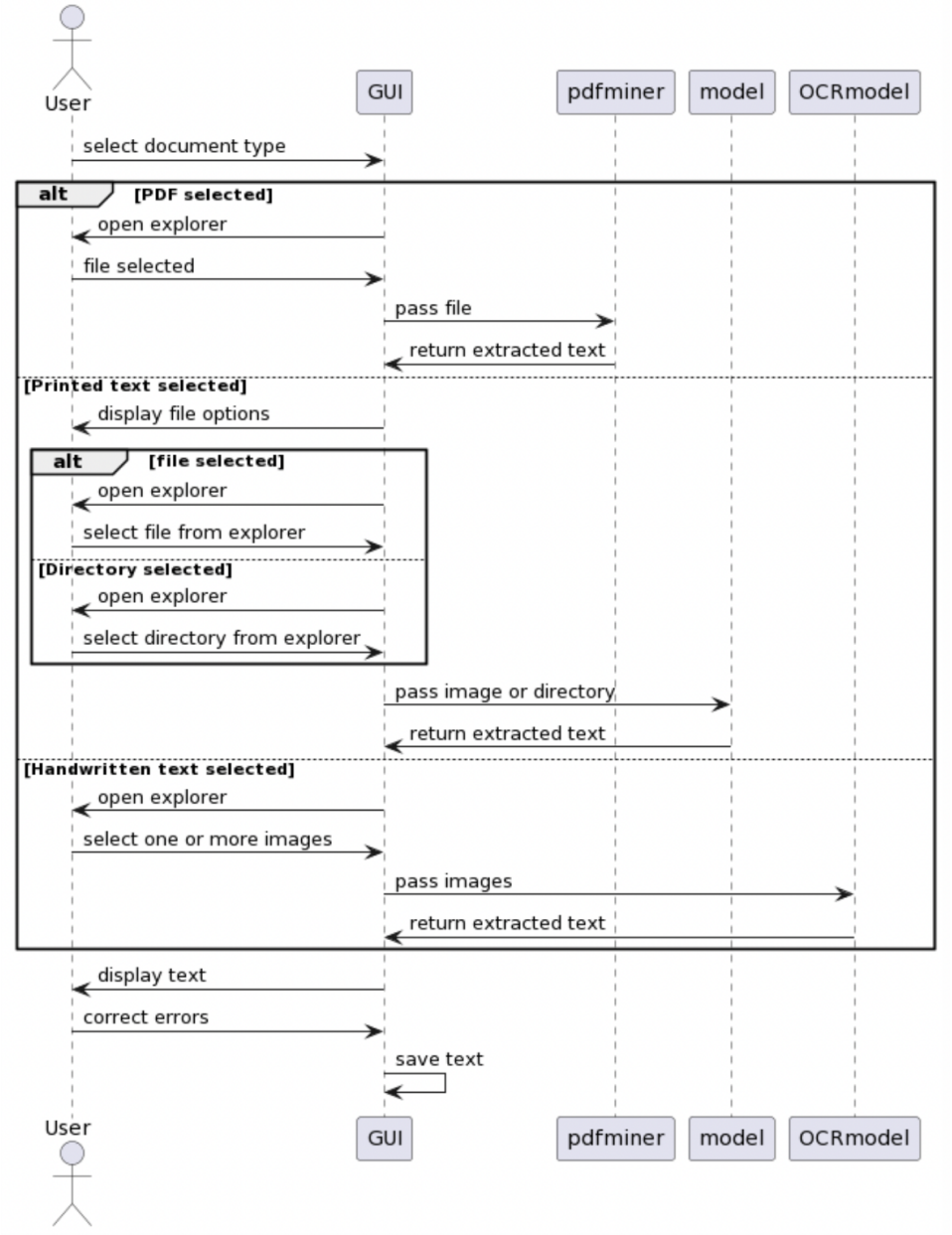


IMAGE THREE: Sequence Diagram for Uploading Documents

3.3.1.2 Modify text

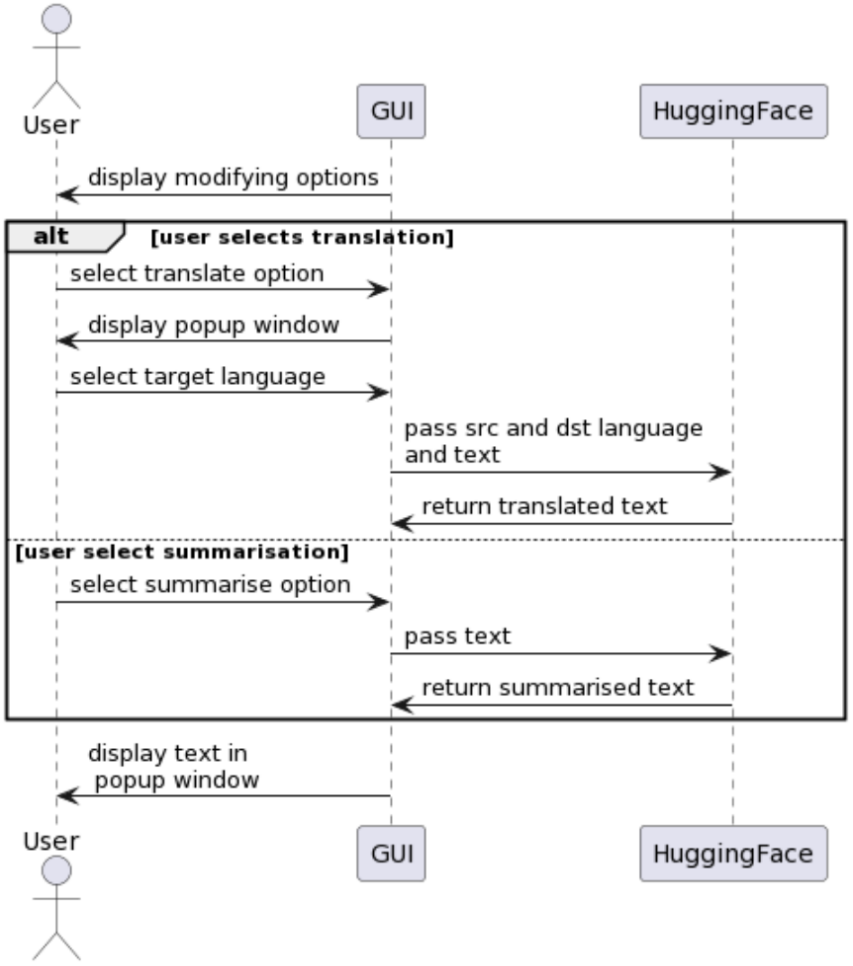


IMAGE FOUR: Sequence Diagram for Modifying the text

3.3.1.3 Download documents

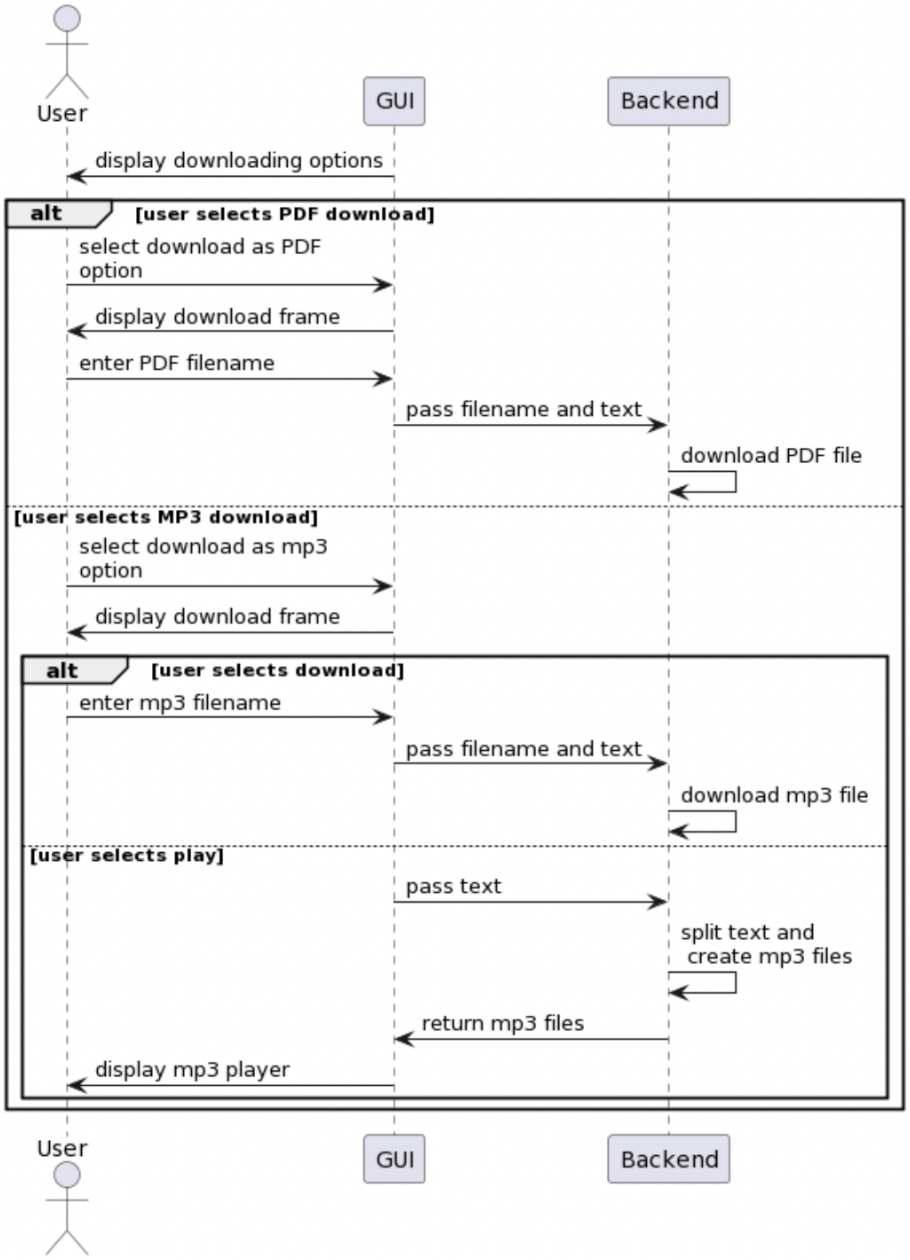


IMAGE FOUR: Sequence Diagram for Downloading the text

5. Implementation

5.1 Front End GUI

For the basic layout of the pages a tutorial [] was followed. This provided the application with boiler-plate code for a main file to call to run the application, and the functionality to switch between frames as though moving through pages of an app.

```
for F in (UploadPage, EditPage, TranslatePage, OptionsPage):
    frame = F(container, self) #create a container for each page
    self.frames[F] = frame # add each frame into the array

    frame.grid(row = 0, column = 0, sticky ="nsew")

self.show_frame(UploadPage)
```

IMAGE : Frames of the Application

Each page is its own class, which are added into an array and are raised when the user visits that frame. The buttons at the bottom of each screen determine the process of moving back to the previous page or onto the next page. The code sample below shows how the application moves between the different frames.

```
# back / next page
previous = Button(btm_frame, text="Back",
            command = lambda : controller.show_frame(edit_page.EditPage))
previous.pack(side='left', padx=8, pady=5)

next = Button(btm_frame, text="Next",
        command = lambda : controller.show_frame(options_page.OptionsPage))
next.pack(side='right', padx=8, pady=5)
```

IMAGE : Functionality of moving between frames

The Tkinter framework has a wide variety of widgets which have been implemented in the application such as Labels, Buttons, Menus, Checkbuttons, and Textboxes. The items displayed on the screen are all tied to the functionality of the app.

5.2 Detection

Detecting the language of the extracted text is crucial to the success of the application, as many of the key functionalities are dependent on knowing the source language. The code

snippet below shows how the detection works. It uses the langdetect library. As langdetect is non-deterministic, the first line forces the detection to be deterministic and prevent the function from returning a different language for the same text. The function detects a language and returns it or returns False if it fails to identify the language.

```
def detection_file(file):
    DetectorFactory.seed = 0 # make the result deterministic
    with open(file, "r") as language_file:
        text = language_file.read()
        try:
            return detect(text)

        except: # if it fails to detect the language
            return False
```

IMAGE : Function to detect the language of a file

5.3 Translation

The translation functionality is implemented using Hugging Face transformers. The translation function is given the source language, the destination language, and the text to translate. The function first finds the correct model and tokeniser to use for the translation. The MarianMT model is used as it allows for fast translations and includes automatic post-editing and grammatical error correction to improve translation quality [4].

```
model_name = f"Helsinki-NLP/opus-mt-{src_lang}-{dst_lang}" # translation model
# get the tokenizer and model for the language pair
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
model.to(device)
```

IMAGE : Initialising the components for the translation model

Once the necessary components are initialised, the text is broken down into batches which are given to the tokeniser and encoded. The model then generates the translation before being decoded by the tokeniser. The translated batch is then appended to the previously translated text, and the total translated text is returned.

5.4 Summarisation

Summarisation is also implemented using Hugging Face transformers. The file of text is passed to a summarisation function which returns a summarised version of the text.

```
if len(re.findall(r'\w+', text)) < 100:
    f.close()
    return text
```

IMAGE : If there are less than 100 words in the text, return text

If the text passed to function is less than 100 words, summarisation is not necessary and the text is returned as it is. Otherwise, the text is passed to a summarisation pipeline. The text is first split into sections of a maximum length of 300 words. The array of sections is then passed to the summariser, where each section will be summarised to a maximum word length of 150, and a minimum of 75 words. The sections are then rejoined to form the summarised text and returned.

```
# initialise the summariser
summariser = pipeline("summarization", model="sshleifer/distilbart-cnn-12-6", device=dev)
# pass the array of sections to the summariser
summarised = summariser(sections, min_length=75, max_length=150, truncation=True)

joined_summ = " ".join([summary['summary_text'] for summary in summarised]) # join all the sections
complete_summ = re.sub(r'\s\.', r'.', joined_summ) # replace " ." with "."
```

IMAGE : Summarisation pipeline

5.5 File Conversion

The final process of the application is converting the text into a new format, be that a PDF file or a mp3 recording. From the GUI, the user selects which output method they would like.

5.5.1 PDF Conversion

Converting the text file into the outputted PDF involves using the library FPDF. The input text to the function ‘convert_to_pdf’ is split on new line characters and copied into the new PDF which is wrapped to ensure that the text does not carry on past the borders.

5.5.2 MP3 Conversion

There are two possible options should the user wish to listen to the text as an audio recording. The user may download the file directly or listen to the audio in the mp3 player. The code snippet below shows how the mp3 is downloaded.

```
# uses gTTS -> Google's text to speech
recording = gTTS(text=text, lang=lang, slow=False)
file_name = os.path.basename(file).replace(".txt", ".mp3")
file_path = os.path.dirname(file) # new file path is constructed

new_file = os.path.join(file_path, file_name)
recording.save(new_file) # the mp3 is downloaded
```

IMAGE : Download mp3

The function uses Google's text to speech capabilities with gTTS. A new file path for the audio file is created and the new file is created as gTTS saves the file. This function to convert a text file to an audio file is also implemented if the user wishes to use the mp3 player. However, the text file is first split into smaller text files before being passed to this function to create the individual audio tracks that are used in the mp3 player.

```
words_lst = words.split() # split the txt file into words
files = [] # array of file paths
chunk = 100 # split into smaller 100 word txt files
for c, i in enumerate(range(0, len(words_lst), chunk)):
        with open("{}/part_{}.txt".format(new_path, c+1), "w") as out:
            out.write(" ".join(words_lst[i:i+chunk])) # write 100 words into new file
            files.append("{}/part_{}.txt".format(new_path, c+1))
```

IMAGE : Split text file into smaller text files

The text file is split into an array of words. New text files are created with a length of 100 words. These are appended to an array of the file paths and returned.

5.6 Single Image OCR

The most basic level of Optical Character Recognition starts with a single image of printed text. This Python script (src/img_ocr.py) takes the path to an image as input, loads the data into OpenCV and performs pytesseract's OCR method on the cv2 image. It then writes the recognised text to a temporary output file "output.txt" which can be used in the next steps of the application.

The image is read in with OpenCV, it is important to note that openCV reads images as BGR channels not RGB and so it requires conversion.

```
img = cv2.imread(img_path)  # load the image from given path
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # conversion to RGB
```

We then perform a number of image preprocessing steps found in src/preprocessing.py.

1. **Adaptive Threshold Binarisation:** This method convert the image to greyscale using Gaussian adaptive threshold binarisation, which finds the local threshold and uses it to decide if the pixel is grey or white (in greyscale).

```
def adaptive_threshold_binarisation(img):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  # convert the image to greyscale

    # calculate the local threshold value using Gaussian weighting and a mean value
    thresh = cv2.adaptiveThreshold(grey, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 21, 4)

    return thresh
```

2. **Deskew:** The deskew method rotates an image back to the upright position. It resizes the image so the edges are not cut-off during rotation

```
angle = determine_skew(grayscale)  # determine the tilt/skew angle of the text

rotated = rotate(image, angle, resize=True) * 255  # transforms the image rotating it by the calculated angle to an
# upright position - set resize to True to avoid cutting off
# part of the image during rotation

return rotated.astype(np.uint8)  # returns an unsigned integer (0-255) representing an image with RGB channels
```

3. **Denoise:** Removes patches of high intensity using non-local means denoising (replaces the pixel intensity value with the mean/average of other pixels.

We run OCR on the processed image to extract the text and write it to the output file, setting the relevant text_model attributes.

5.7 Image Folder

Similar to what we have seen above, this Python script (src/folder_img_ocr.py) also performs OCR on images of printed text, however, this time it loops through a folder of such images, combining their respective outputs together into a single output file, which is passed forward to the rest of the application's functions.

In this case, the function takes in the path to the directory/folder passed from the upload section of the GUI and loops through the contents. We must be sure to append the image name onto the path to extract the text from each file individually.

```
for image in os.listdir(os.path.join(folder_path)):
    img = Image.open(os.path.join(os.path.join(folder_path), image))  # access the image
```

Notably, missing from this step of the OCR model is the image preprocessing methods. This is simply because of the additional performance costs associated with running these functions over many image files and the serious hardware limitations of the machines used to develop the application. However, these preprocessing steps could still be easily included into the folder OCR program.

```
# image preprocessing methods to improve OCR accuracy
img = preprocessing.deskew(img)  # deskews an image with a tilt
img = preprocessing.adaptive_threshold_binarisation(img)  # binarise the image using gaussian adaptive threshold
img = preprocessing.denoise(img)  # image denoising

# use Tesseract to OCR the image
text = pytesseract.image_to_string(img)
```

5.7.1 OCR on PDFs

The same program is used when performing OCR on PDF files inputted by the user. When a PDF is inputted each page is converted into its own image (.jpg) file.

```
def open_pdf():
    pdf_path = askopenfilename(filetypes=[('PDFs', '*pdf')])

    if pdf_path is not None:

        pdf2jpg.convert_pdf2jpg(pdf_path, "./", pages="ALL")
```

All of the images (pages) are stored in a temporary folder (*PDF NAME*.pdf_dir) which is passed to the folder OCR script to extract the text. The temporary folder is then deleted as it is no longer necessary.

```
def pdf_ocr(self, btm_frame, controller, files, confirm_btn):
    folders_ocr(text_model.get_dir_path())

    rmtree(text_model.get_dir_path())  # remove temporary folder of images from pdf
```

5.8 Handwriting Text Recognition (HTR)

The HTR model is trained on the IAM Handwriting Database (https://fki.tic.heia-fr.ch/databases/iam-handwriting-database), which contains 13,353 text lines and 115,320 words handwritten in English. The model is pre-trained to recognise short handwritten sentences and stored in the src/htr/model directory.

5.8.1 Load IAM HTR Data

As mentioned, the HTR model is trained on the IAM Database. The database comes with thousands of words which must be loaded and processed before being used to generate the model and a text file containing a summary of the image data.

To load in this data, we must first extract the necessary data from this summary. The summary file is clearly divided into sections, which makes it easy to acquire the information separately using a series of split() calls.

```
words_file = open("./words.txt")  # summary of word data in IAM Dataset
chars = set()
broken_words = ['a01-117-05-02', 'r06-022-03-05']  # ids of known broken word images

for line in words_file:

    # skip empty lines and comments
    line = line.strip()

    if not line or line[0] == '#':
        continue

    # each line consists of 9 space delimited sections
    split_line = line.split(' ')
    assert len(split_line) >= 9  # check each split line contains at least 9 parts

    # split filename into different parts
    # part1/part1-part2/part1-part2-part3.png
    filename_split = split_line[0].split('-')  # split a01-000u-00-00
    subdir1 = filename_split[0]  # a01
    subdir2 = f'{filename_split[0]}-{filename_split[1]}'  # a01-000u
    file_id = split_line[0] + ".png"  # Image in subdirectories: a01-000u-00-00.png
    filename = data_dir / "img" / subdir1 / subdir2 / file_id  # data/img/a01/a01-000u/a01-000u-00-00

    # skip over broken word images
    if split_line[0] in broken_words:
        print("Skipping known broken image: ", filename)
        continue

    gt_text = ' '.join(split_line[8:])  # ground text of an image start at col 9 (can continue if considering
```

We begin by looping through the lines of the summary data and ignoring any commented lines (“#”) and known broken images (seen in the list above). We then split each line along whitespaces to get the different bits of information (e.g. image ID, path to image, image directory and ground truth texts etc.). After this, we can split the loaded data into a training and validation set (95%:5%).

```
    # divide data into training (95%) and validation (5%) sets
    split_index = int(data_split * len(self.samples))
    self.train_samples = self.samples[:split_index]
    self.validation_samples = self.samples[split_index:]

    # populate lists with words
    self.train_words = [x.gt_text for x in self.train_samples]
    self.validation_words = [x.gt_text for x in self.validation_samples]
```

The data loader class also has a number of simpler help functions which are called later, including get image, has next (element) and get iterator information (fetches the current training batch and the total number of batches).

5.8.2 HTR Model

Htr_model.py is actually responsible for creating the HTR model. The HTR model consists of a number of Convolution Neural Network layers (CNN), Recurrent Neural Network (RNN) layers and a Connectionist Temporal Classification (CTC) decoder layer.

The model has 5 CNN layers that are designed to extract the feature information from the image. Each CNN layer applies a filter kernel of various sizes (5x5 for the first 2 layers, 3x3 for the final 3 layers).

```
kernel_vals = [5, 5, 3, 3, 3]  # 5x5 kernel for first 2 layers, 3x3 for final 3 layers
```

CNN

We use tensorflow to create the kernel for the convolution process. The convolution is given a stride of 1 so that it considers each pixel and uses padding so we do not decrease the resolution for the output of the CNN layers and then normalised.

```
for i in range(num_layers):  # loop over all CNN layers - extract relevant features

    # variable Tensor - 4D for 2D convolution operation
    # create a kernel of kernal_vals[i] x kernal_vals[i]
    kernel = tf.Variable(
        tf.random.truncated_normal([kernel_vals[i], kernel_vals[i], feature_vals[i],
                                    feature_vals[i + 1]], stddev=0.1))

    # 2D convolution using same padding + kernel -> set stride of sliding window to 1
    convolution = tf.nn.conv2d(input=pool, filters=kernel, padding='SAME', strides=(1, 1, 1, 1))
    norm_conv = tf.compat.v1.layers.batch_normalization(convolution, training=self.is_train)  # normalised conv
```

We use the non-linear RELU activation function on the normalised convolved image which returns 0 for negative values and the value itself if it is positive. Finally, we use pooling to summarise image regions to output a downsized version of the image.

RNN

We implemented a bidirectional Long Short-Term Memory (LSTM) RNN. The LSTM RNN is capable of propagating information through longer distances providing a more robust output than a basic RNN.

```
# Bidirectional RNN - forward + backwards (2 output seqs 32x256)
(fw, bw), _ = tf.compat.v1.nn.bidirectional_dynamic_rnn(cell_fw=stacked, cell_bw=stacked,
                                                        inputs=rnn_3d, dtype=rnn_3d.dtype)

# concatenate forward + backwards cells along the 2nd dimension -> expand dimension @ axis 2
concat = tf.expand_dims(tf.concat([fw, bw], 2), 2)  # outputs seq of size 32x512
```

CTC

The CTC decoder is passed the output of the RNN layer as input. We take the ground truth texts as a sparse tensor and compute the loss value.

```
# ground truth texts encoded as sparse tensor -> 3 separate dense tensors (indices, values, dense_shape)
self.gt = tf.SparseTensor(tf.compat.v1.placeholder(tf.int64, shape=[None, 2]),
                          tf.compat.v1.placeholder(tf.int32, [None]),
                          tf.compat.v1.placeholder(tf.int64, [2]))

self.seq_len = tf.compat.v1.placeholder(tf.int32, [None])  # input seq len (passed into CTC loss and decoding)

# calculate loss for batch -> find mean of elements across dimensions of ctc loss tensor
self.loss = tf.reduce_mean(
    input_tensor=tf.compat.v1.nn.ctc_loss(labels=self.gt, inputs=self.ctc_3d,
                                          sequence_length=self.seq_len,
                                          ctc_merge_repeated=True))
```

The option of 3 different decoders is also given (BestPath, BeamSearch and WordBeamSearch). BestPath and BeamSearch are implemented using Tensorflow while WordBeamSearch is implemented from a 3rd party git repository.

5.8.3 Main_htr.py

The main_htr.py acts as its own interface with the rest of the components making up the HTR model process. From here, we can train or validate the NN or call inference on an image with handwriting to extract the text from the image.

6. Problems Solved

6. Translation performance

A significant issue we encountered was the poor initial performance of the translation implemented with Hugging Face transformers. The translation function took a significant amount of time and struggled to translate large pieces of text. When given an A4 page of text, the initial time taken to translate the page was 15 seconds. However, the function only translated the first 171 words of the translation from 488 words in total.

This issue was remedied by splitting the text into batches and running the translation on the GPU. This improved the translation performance to 11.7 seconds, and the whole page was translated. The improved function translated almost three times as much text as the original version and completed it in 3.3 seconds less.

6. Splitting mp3 recording

For the mp3 player, the audio recording is split into smaller tracks, to make it easier and more efficient to use. The original plan was to use the ‘pydub’ library to achieve this. However, it proved very difficult to install this library and its dependencies so an alternative method was used to split the recordings.

Our solution to this problem was to first split the text file which the audio is created from. This is incredibly straightforward as it can be split simply on how many words should be in each file. These smaller text files are then individually converted into audio files which are stored in a folder together. Splitting the text file first ensures that the audio files are all relatively the same length and will not end in the middle of a word.

7. Future Work

7.1 Expand Languages

The application currently only supports translation in English, French, and German. Future work could involve expanding on these languages and potentially support non-latin alphabet languages. The OCR model for images of handwritten text is trained to only recognise and extract English text. This could also be expanded on to support languages other than English.

7.2 Improve HTR model

The current HTR model is limited to recognising text similar or identical to that of text seen in the IAM database. HTR is still an area of research in computer science and as such is still limited in its success, especially given the added limitations of a student project. That being said the model could be improved by running training the NN for longer or theoretically by expanding the types of text it is trained on beyond the IAM database.

7.3 Improving Folder OCR

As mentioned above no preprocessing steps were taken when performing OCR on a folder of images, due to the added complexity and resource requirements of performing such processes on a large number of files. This could easily be improved upon in future with better hardware and more time.

8. References

[1] CPD (2022) *The benefits of audio learning*, *The CPD Certification Service*. The CPD Certification Service. Available at: https://cpduk.co.uk/news/the-benefits-of-audio-learning#:~:text=By%20listening%20to%20audio%20recordings,be%20in%20a%20physical%20location. (Accessed: May 4, 2023).

[2] A. Rossiter, A. Nortcliffe, A. Griffin & A. Middleton (2009) Using student generated audio to enhance learning, Engineering Education, 4:2, 52-61, DOI: 10.11120/ened.2009.04020052

[3] Fleck, R. (2021) *10 fundamental UI design principles you need to know*, *Dribbble*. Available at: https://dribbble.com/resources/ui-design-principles (Accessed: May 6, 2023).

[4] Sharma, N. (2023) *Hugging face pre-trained models: Find the best one for your task*, *neptune.ai*. Available at: https://neptune.ai/blog/hugging-face-pre-trained-models-find-the-best (Accessed: May 6, 2023).