Assignment Cover Letter

(Individual Work)

Student Information:	Surname	Given Names	Student ID Number
	Wijadi	Jason Jeremy	2440077301

Course Code	: COMP 6056	Course Name	: Program Design Methods
Class	: L1BC	Name of Lecturer(s)	: Ida Bagus Kerthyayana Manuaba
Major	: Computer Science		

Title of Assignment: Crude OCR
(if any)

Type of Assignment : Final Project

Submission Pattern

Due Date : 10/01/2021 **Submission Date : 10/01/2021**

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.

"Crude OCR"

Name : Jason Jeremy Wijadi

ID : 2440077301

Project Specification

The objective of this program is to analyze written text and convert it to a format which can be directly copied for further use digitally. The program will be able to accept image inputs in the form of a JPG or PNG file, as well as a frame capture from the user's webcam. An option to test the AI and its prediction capabilities will also be able to be selected by the user in the form of a drawing canvas. In addition to the final copyable text, the user will also have the option to have the text be read aloud through text-to-speech.

1. Program **input**:
    a. JPG or PNG file
    b. JPG file from the webcam
    c. PNG file from the drawing canvas
2. Program **output**:
    a. Selectable text from AI prediction of images
    b. Audio playthrough of AI prediction
    c. Pop up message box from AI prediction of drawing

Third-party libraries / modules used:

1. TensorFlow
    - Making deep learning AI model using neural networks
    - Loading in dataset
2. NumPy
    - Dataset processing (AI training and testing)
    - Final image processing (image for predictions)
3. h5py
    - Reading and writing AI model (model uses .h5 file format)
4. Matplotlib
    - Visualizing the evaluation of AI model with dataset
5. TkInter
    - Program main GUI
6. OpenCV
    - Webcam operation
    - Input image processing (Image for predictions)
7. Pillows
    - Converting images to TkInter displayable format
    - Grabbing drawing canvas image
8. Imutils
    - Part of input image processing (image for predictions)
9. gTTS
    - Creating audio file from prediction text
10. playsound
    - Playing audio file from gTTS

Solution Design

The GUI for the program consists of a single window. Each different interface will overwrite the previously displayed one.
The list of the interfaces are as follows:

1. Main menu
2. Webcam frame
3. File frame
4. Testing frame
5. Result frame

Main Menu

The user starts the program on the main menu. There are 3 buttons on this frame: Start Webcam, Open File, and Test the AI!. Each of these buttons will lead the user to the linked interface that the button is associated with. The buttons themselves also reflect the 3 different ways that the user can give input to the program.

Webcam Frame

The webcam frame hosts the webcam input from the user. There is a button for the user to switch the current activated webcam to the next available one if the user has more than one webcam connected to their machine. Before clicking the “Confirm?” button to capture webcam input, the user has to select between 3 options to determine the type of text that they’re trying to input. More info about this will be presented later on in the report. There is also a back button, which will lead the user back to the main menu.

File Frame

The file frame hosts the file input from the user. Before the frame is loaded, a file dialog is shown for the user to choose an image that they want input into the program. The chosen image will then be displayed on the frame. An option to immediately process the image is presented in the form of a “Confirm?” button, though the user also has an option to add more images to the prediction through the “Add More Images” button. Like the process in the webcam frame, the user has to select between the type of text that they’re trying to input for each image. There is also a “Delete Last Image” button, which clears the previous image that was selected by the user if they accidentally selected an unintended image. There is also a back button, which will lead the user back to the main menu.

<u>Testing Frame</u>
The testing frame hosts the drawing canvas which the user can draw on using their mouse input. The mouse left click is associated with drawing black lines, while the mouse right click is associated with deleting drawn lines. Unlike the webcam and file frame, the testing frame immediately gives the prediction result when clicking the “Confirm?” button in the form of a pop up message box. The user will then see the prediction result and will have the option to either draw again, or go back to the main menu. As usual, there is also a back button that leads the user back to the main menu.

<u>Result Frame</u>
This result frame is where the results for the prediction of the images that come from the webcam and file frame lead to. A text box is present which will have the predicted text inside it. The user can then copy the text from the text box and continue on with their work. The “Speech” button will attempt to read from the text and play the audio back to the user. As usual, there is also a back button that leads the user back to the main menu.

Flowcharts

Main Menu (0)

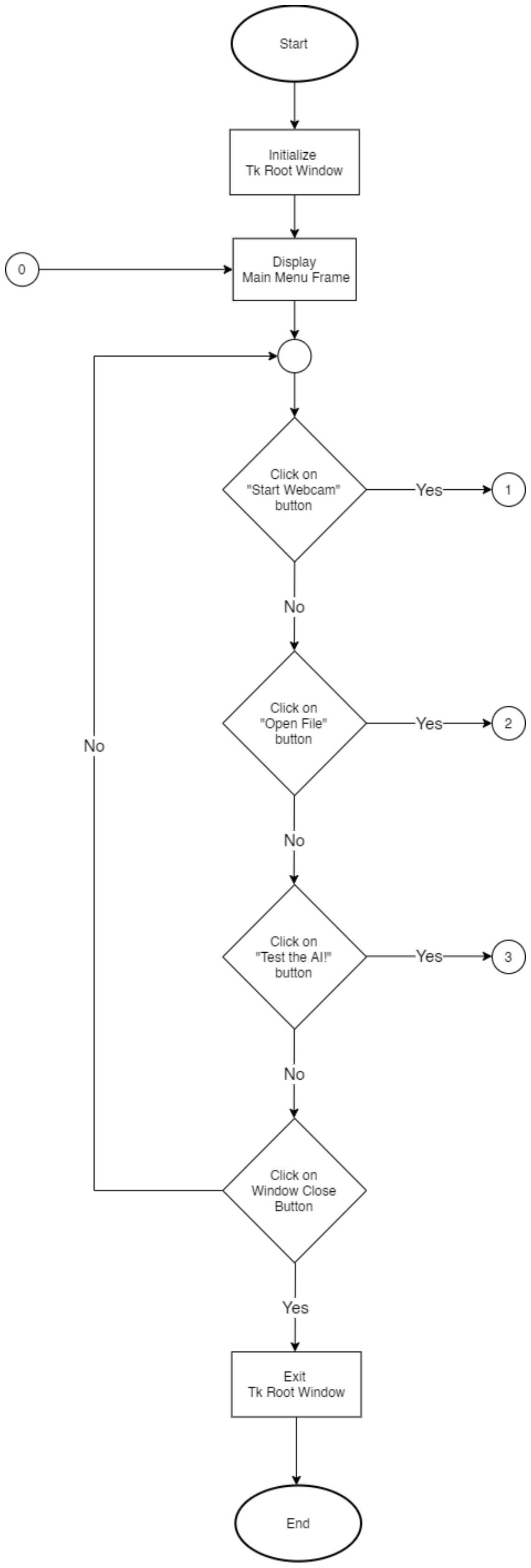

Start
Initialize Tk Root Window
0
Display Main Menu Frame
Click on "Start Webcam" button
Yes
1
No
Click on "Open File" button
Yes
2
No
No
Click on "Test the AI!" button
Yes
3
No
Click on Window Close Button
Yes
Exit Tk Root Window
End

Fig. 1: Main Menu Flowchart

Webcam Frame (1)

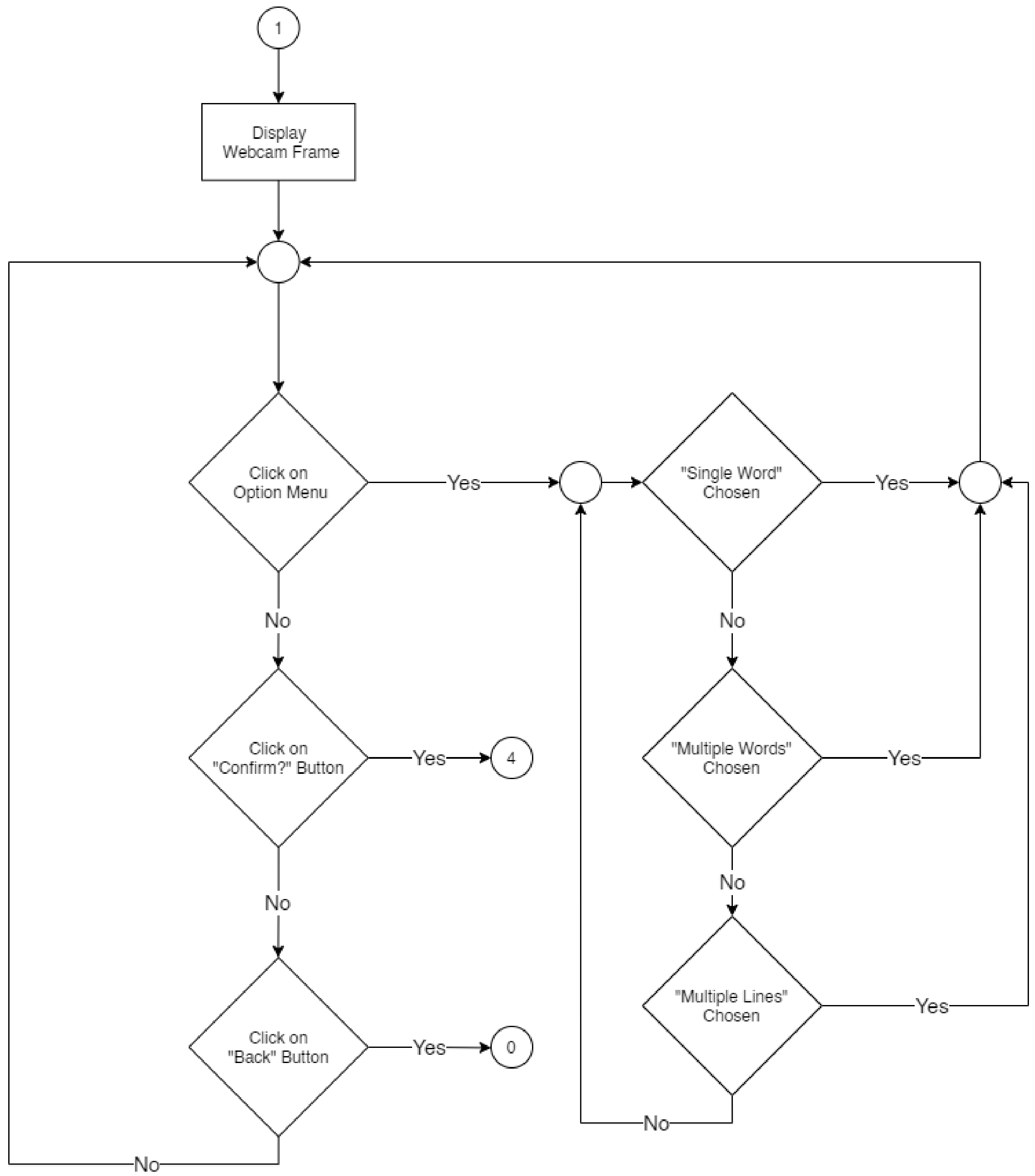


Fig. 2: Webcam Frame Flowchart

File Frame (2)

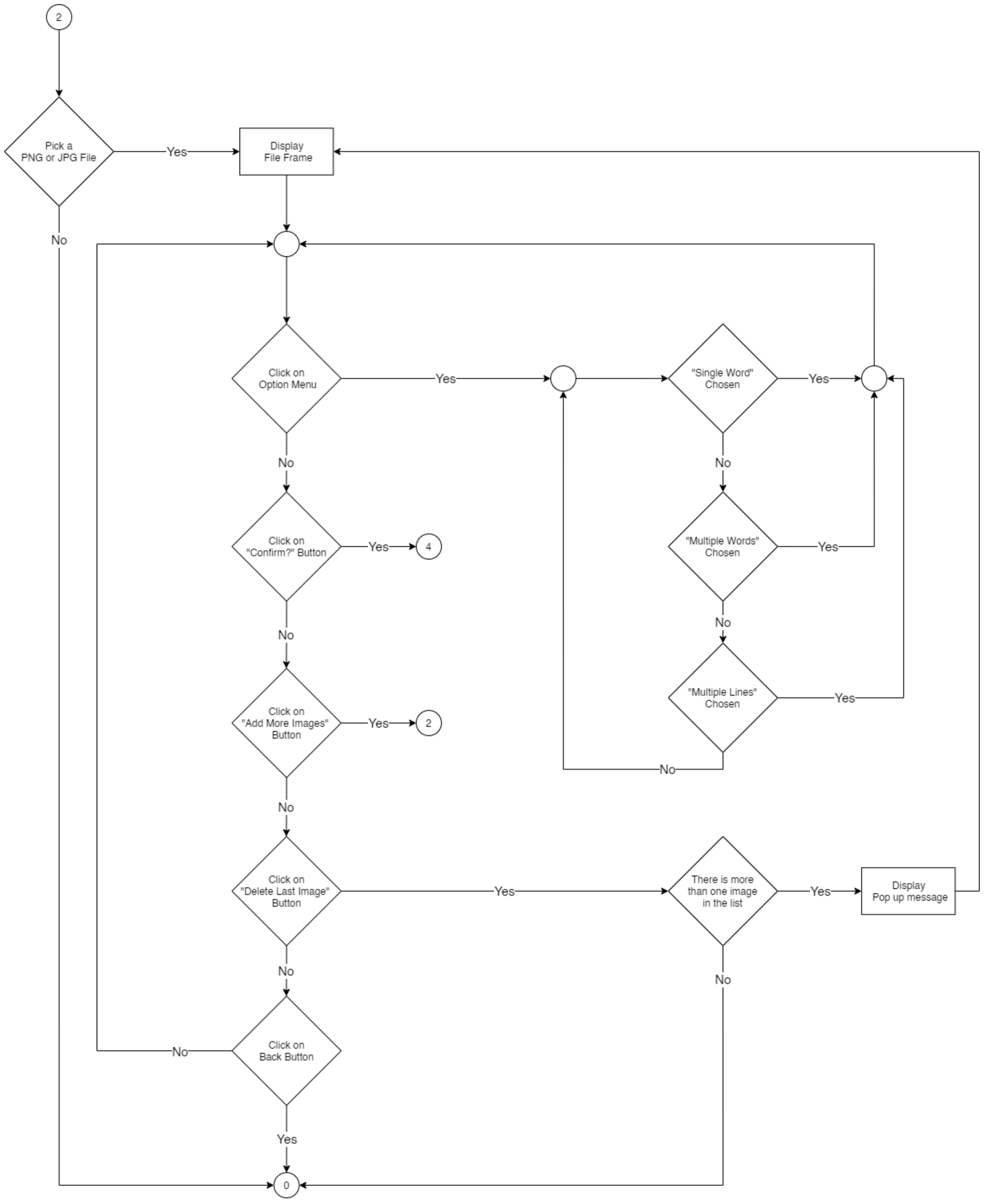


Fig. 3: File Frame Flowchart

Testing Frame (3)

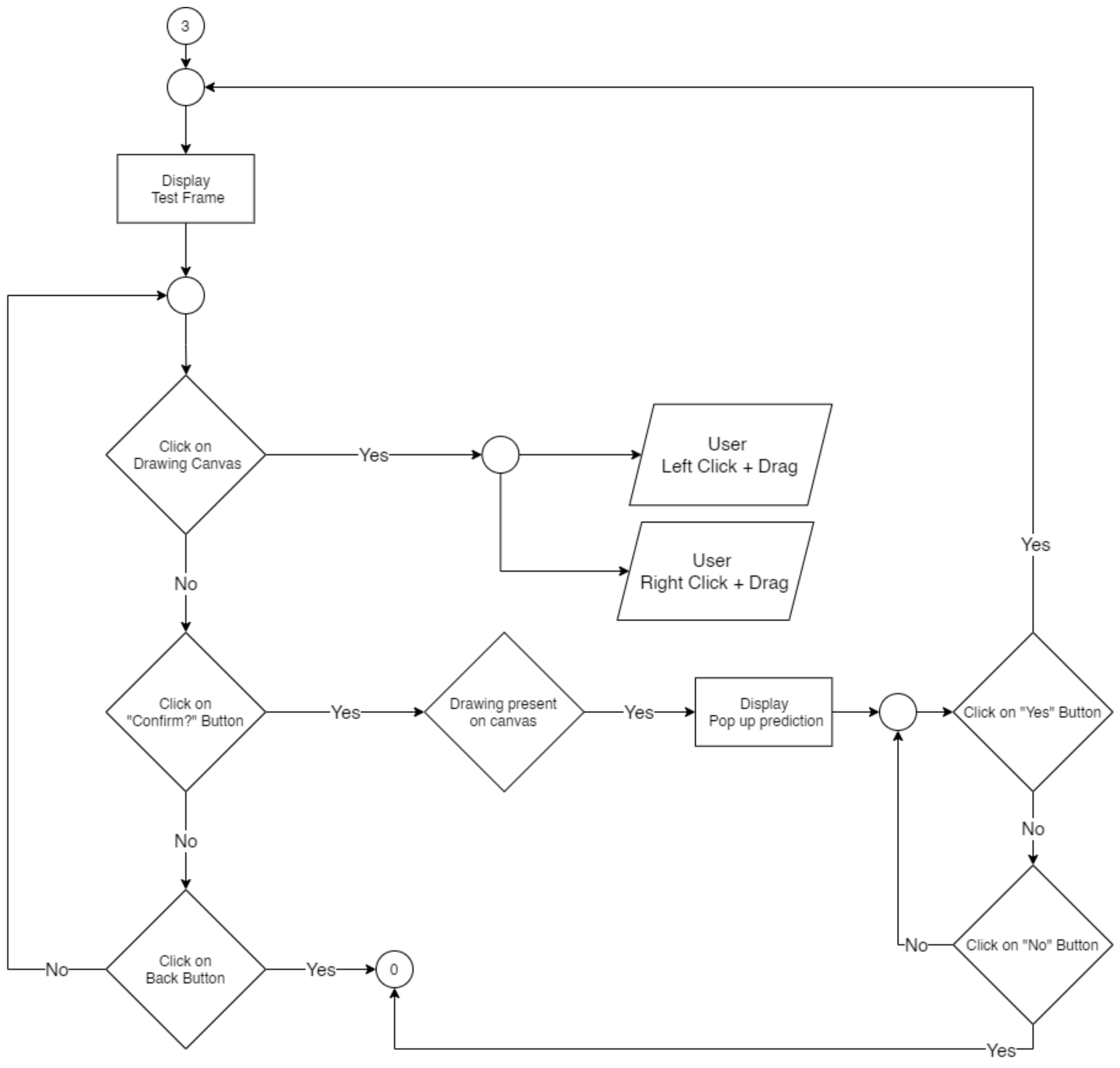


Fig. 4: Testing Frame Flowchart

Result Frame (4)

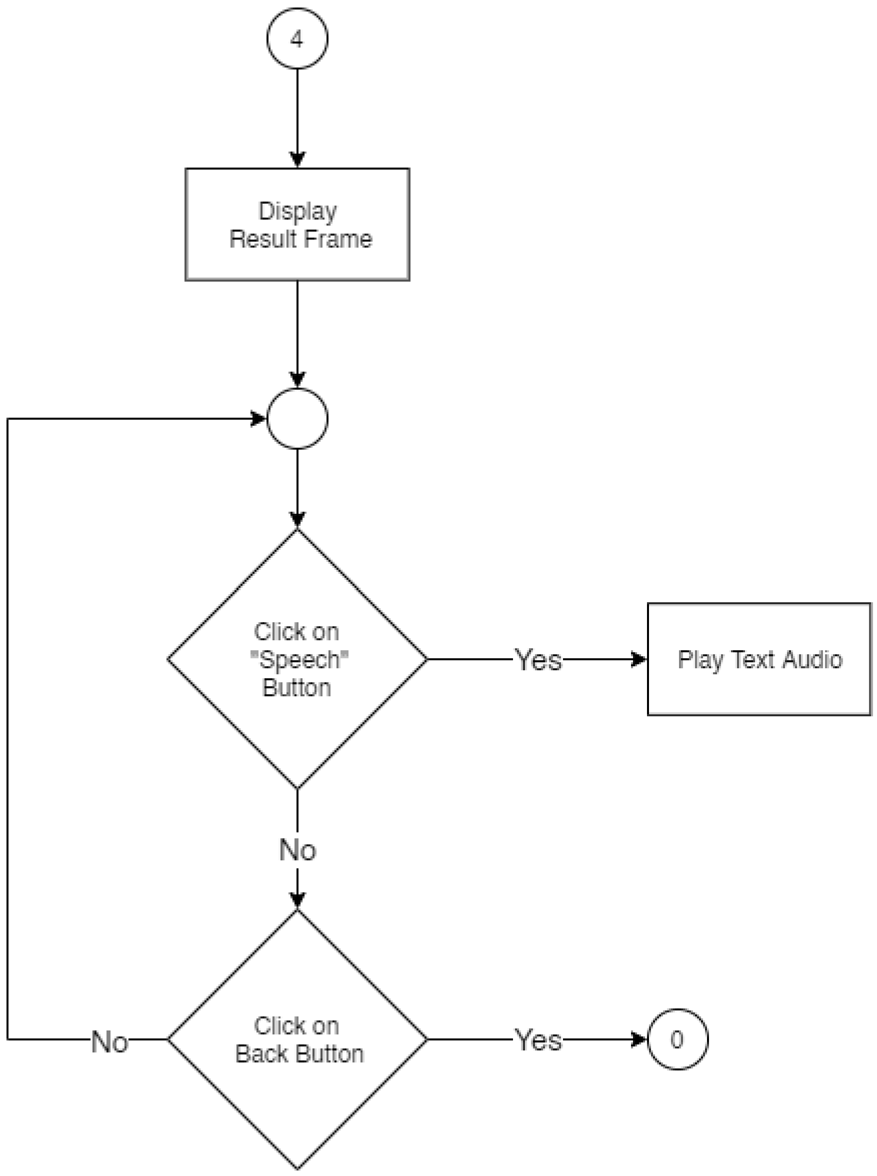


Fig. 5: Testing Frame Flowchart

Program Notes

For webcam and file input, users must choose the correct text type for their image from the option menu. Otherwise, the AI model won't be able to predict the image properly.
Punctuations are not yet supported.

For Mac Users

1. The playsound module currently doesn't work by itself. To get around this, do

```
------------------------------
pip install PyObjC
------------------------------
```

2. **The current way that the testFrame works to grab an image of the canvas is broken on Mac. There is currently no workaround for that problem hence, testFrame doesn't work on Mac.**

3. When opening the webcam, macOS tends to ask the user to allow camera permission for the app that's opening it. In the case that it doesn't, the app needs to be run with **sudo** permissions.
An example with Visual Studio Code

```
--------------------------------------------------------------------------------------------------------
sudo /Applications/Visual\ Studio\ Code.app/Contents/MacOS/Electron
--------------------------------------------------------------------------------------------------------
```

For Model Training / Testing

For NVIDIA GPU Owners, follow the instructions in this link for faster model processing

The dataset used for the current model, and how it's set up, is in data/data.py

Data is saved in .pickle format for quicker access during model training

Follow the comments and directions in data/README.md and model.py to test new / existing models

Program Implementation

I'm breaking this part down into multiple parts, according to the chronological order of how I implemented everything.

1. Preparing the Dataset and Creating AI Model
2. Creating the TkInter Interface Structure
3. Webcam Frame
4. File Frame
5. Testing Frame
6. Image Processing Implementation
7. Result Frame

Preparing the Dataset and Creating AI Model

General Goal:

- Getting an image through to the AI Model for it to predict and give back result(s)
- Produce the result(s) in an easy to read format
- The result(s) have to be "accurate enough"

Normally, OCR, short for Optical Character Recognition, implementations are able to detect and accurately reproduce text through the use of a large library of fonts that have already been premade. Once it tries to analyze something that it doesn't have in its database, such as handwritten text, it tends to fail heavily on reproducing the text and its characters. Therefore, instead of referencing from a large library of fonts, machine learning can be implemented into an OCR system for it to be able to accurately predict handwritten text. This is called ICR, short for Intelligent Character Recognition. Even though it has a specific term, I'll still be using OCR for the rest of the explanation as ICR itself is a subsection in OCR.

For this project's OCR implementation, I'll be creating an AI Model that trains from a large dataset of handwritten characters consisting of digits (0 - 9), uppercase letters (A - Z), and lowercase letters (a - z).

AI Model

The basic idea of machine learning is matrix transformations. In this project's case, an image is broken down from 2-dimensions (e.g. resolution of 28x28), to 1-dimension (e.g. 28x28 -> 784). All of these pixels represent the different parts and features of an image. In the case of a character, it would mean its curves and lines. From this 1-dimensional array of values (pixels are represented in values from 0 - 255 with 0 being pure black and 255 being pure white), the AI model takes each value and represents it as its own neuron / cell. From here, computations are done depending on the type of neural network and the layers that comprise it which will then output a prediction. In this project's case, the prediction will comprise of 62 neurons which cover all the characters that were mentioned previously. Predictions may well be very inaccurate the first time around but, as the model gets continually trained and "learns from its mistakes", accuracy should increase.

However, before even attempting to create an AI model that trains from a set of images, we have to prepare the images first.

Dataset Preparation

Usually, OCR implementations using machine learning use the MNIST (Modified National Institute of Standards and Technology) dataset, which covers a large set of 60,000 images to train from and 10,000 images to test from. However, because we also want to cover uppercase and lowercase letters, we use the EMNIST dataset instead, which extends the original MNIST dataset and adds the characters that we need for training.

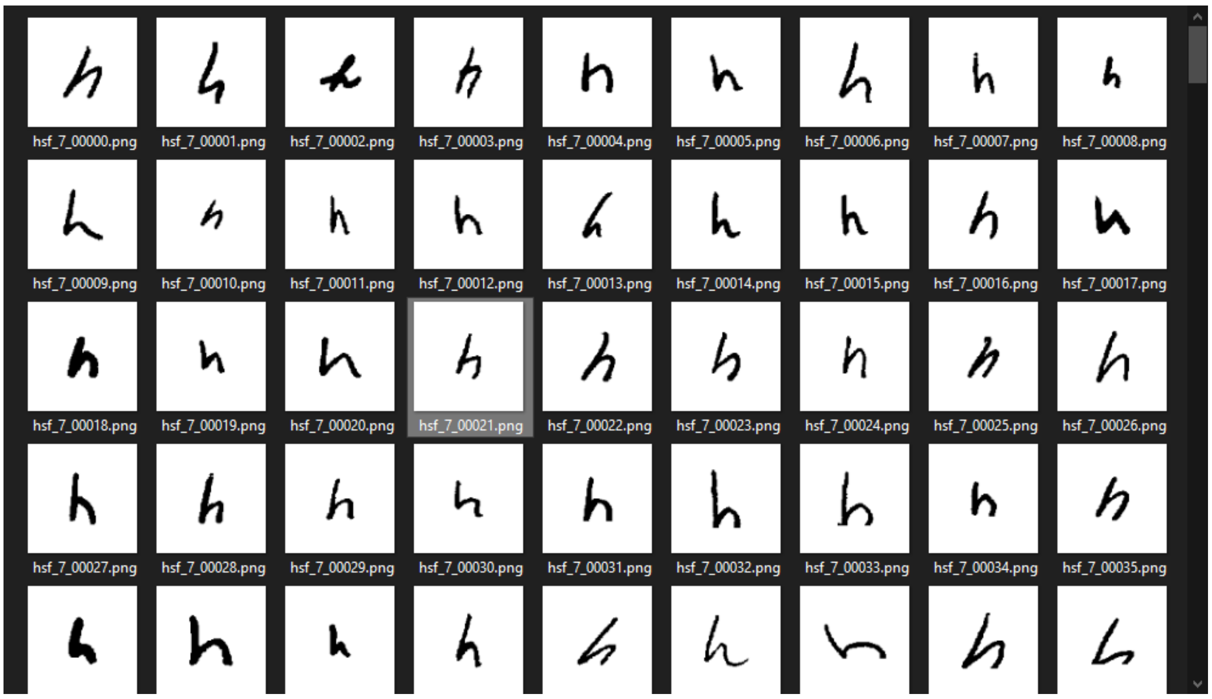
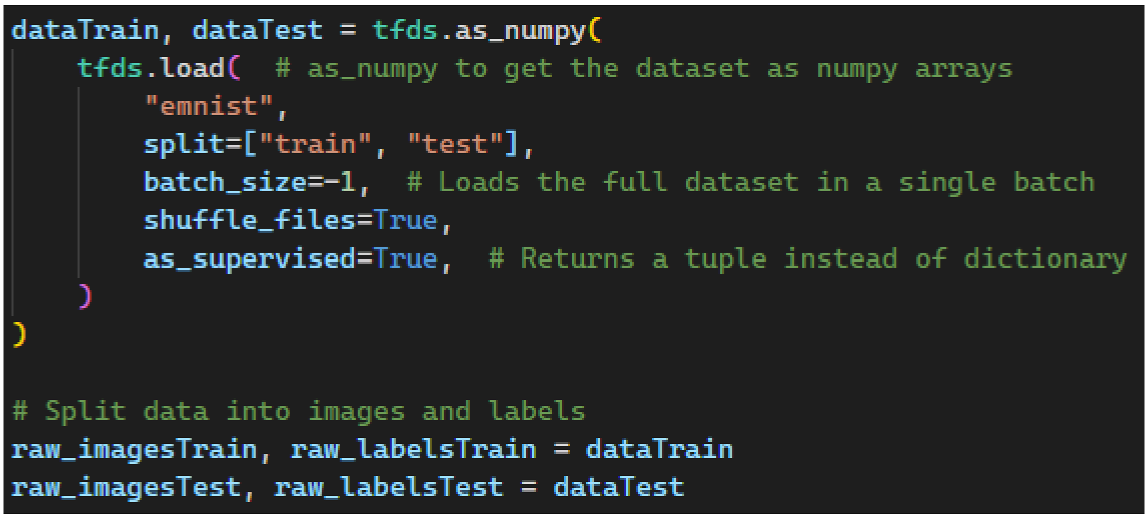


Fig. 6: An example folder from the EMNIST dataset

At its raw state, the images inside the dataset are 128x128 pixels large and are in a folder format that's quite hard to read. Thankfully, as we are using TensorFlow for this project, the TensorFlow Datasets library includes the EMNIST dataset in a much easier format to read and load from. They are also smaller in size with each image being 28x28 pixels large. This will make it a lot faster to process later on.

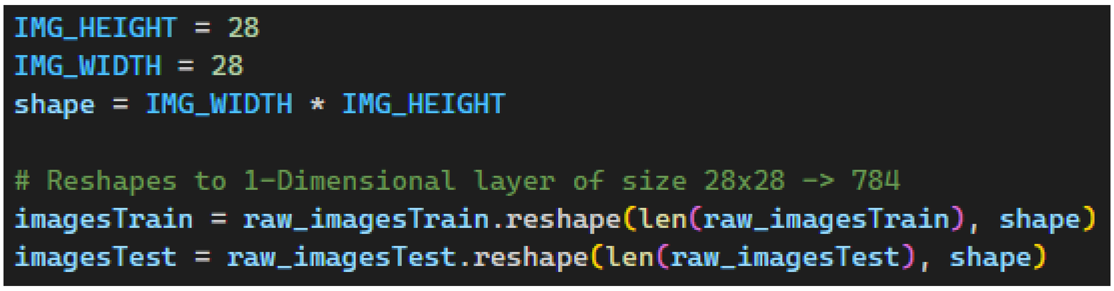
There are a total of 697,932 images for the AI to train with and 116,323 images for testing. Unfortunately, the EMNIST dataset from the TensorFlow Datasets library are currently inverted horizontally and are rotated 90 degrees anti-clockwise. So before we can push it through for AI model training, some preparations need to be done.



```python
dataTrain, dataTest = tfds.as_numpy(
    tfds.load(  # as_numpy to get the dataset as numpy arrays
        "emnist",
        split=["train", "test"],
        batch_size=-1,  # Loads the full dataset in a single batch
        shuffle_files=True,
        as_supervised=True,  # Returns a tuple instead of dictionary
    )
)

# Split data into images and labels
raw_imagesTrain, raw_labelsTrain = dataTrain
raw_imagesTest, raw_labelsTest = dataTest
```

Fig. 7: TensorFlow Dataset (as tfds) EMNIST loading

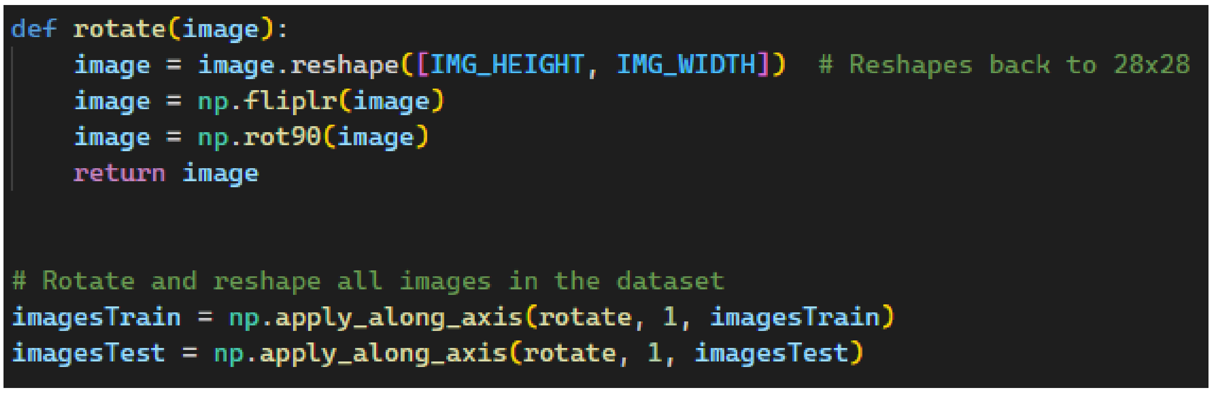
Firstly, we run the command to load the TensorFlow dataset, "tfds.load", and specify some arguments. "emnist" to get the EMNIST dataset, "split" to split the dataset into 2 parts, "batch_size" of -1 to include the full dataset, "shuffle_files" set to True so that there's some randomness in the file structure, and "as_supervised" set to True for it to load the dataset as a tuple instead of a dictionary. We then split the data into 2 variables, dataTrain and dataTest, and pass them on using "tfds.as_numpy" to convert the dataset as numpy arrays. Once that's done, we split the training and testing data again to 2 variables each, raw_images and raw_labels. The raw_images consists of the images, while raw_labels is the appropriate character that corresponds to the image (e.g. an image of character "A" is a label "A").



```python
IMG_HEIGHT = 28
IMG_WIDTH = 28
shape = IMG_WIDTH * IMG_HEIGHT

# Reshapes to 1-Dimensional layer of size 28x28 -> 784
imagesTrain = raw_imagesTrain.reshape(len(raw_imagesTrain), shape)
imagesTest = raw_imagesTest.reshape(len(raw_imagesTest), shape)
```

Fig. 8: Reshaping EMNIST dataset images

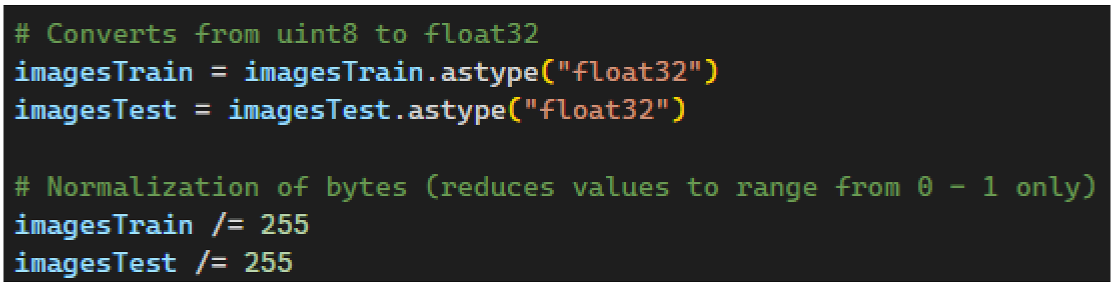
Once that's done, we convert the shape of the images into a 1-dimensional layer of size 784 for our reshape function after this.



```python
def rotate(image):
    image = image.reshape([IMG_HEIGHT, IMG_WIDTH])  # Reshapes back to 28x28
    image = np.fliplr(image)
    image = np.rot90(image)
    return image


# Rotate and reshape all images in the dataset
imagesTrain = np.apply_along_axis(rotate, 1, imagesTrain)
imagesTest = np.apply_along_axis(rotate, 1, imagesTest)
```

Fig. 9: Fixing the orientation of EMNIST dataset images

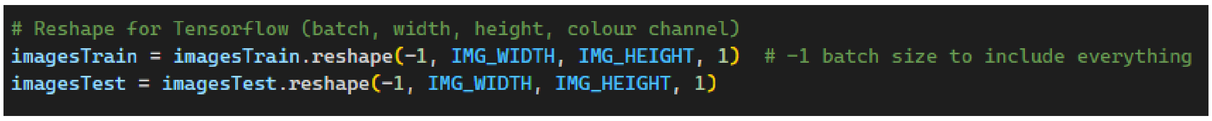
Using the NumPy library, we reshape the images back to 28x28 while flipping the images horizontally and rotating them by 90 degrees to get the correct orientation. We then apply that rotate function to every image.



```python
# Converts from uint8 to float32
imagesTrain = imagesTrain.astype("float32")
imagesTest = imagesTest.astype("float32")

# Normalization of bytes (reduces values to range from 0 - 1 only)
imagesTrain /= 255
imagesTest /= 255
```

Fig. 10: Normalization of bytes

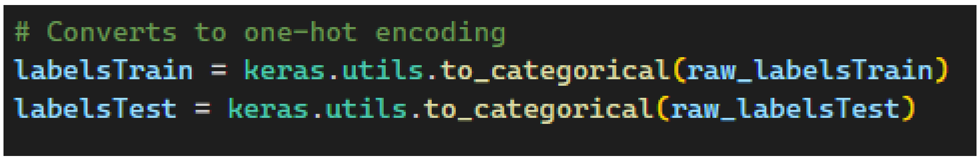
As the images are in their raw state (values of 0 - 255), we want to binarize them into values of 0 - 1 instead to make the processing easier. To do that, we convert them to values of float32 (supports decimal points), and divide all of them by 255.



```python
# Reshape for Tensorflow (batch, width, height, colour channel)
imagesTrain = imagesTrain.reshape(-1, IMG_WIDTH, IMG_HEIGHT, 1)  # -1 batch size to include everything
imagesTest = imagesTest.reshape(-1, IMG_WIDTH, IMG_HEIGHT, 1)
```

Fig. 10: Final reshape

Finally, we reshape the images again for them to include 1 colour channel. This 1 colour channel makes them appear grayscale, which is what we're going to input to the AI model for training and for predictions later on.



```python
# Converts to one-hot encoding
labelsTrain = keras.utils.to_categorical(raw_labelsTrain)
labelsTest = keras.utils.to_categorical(raw_labelsTest)
```

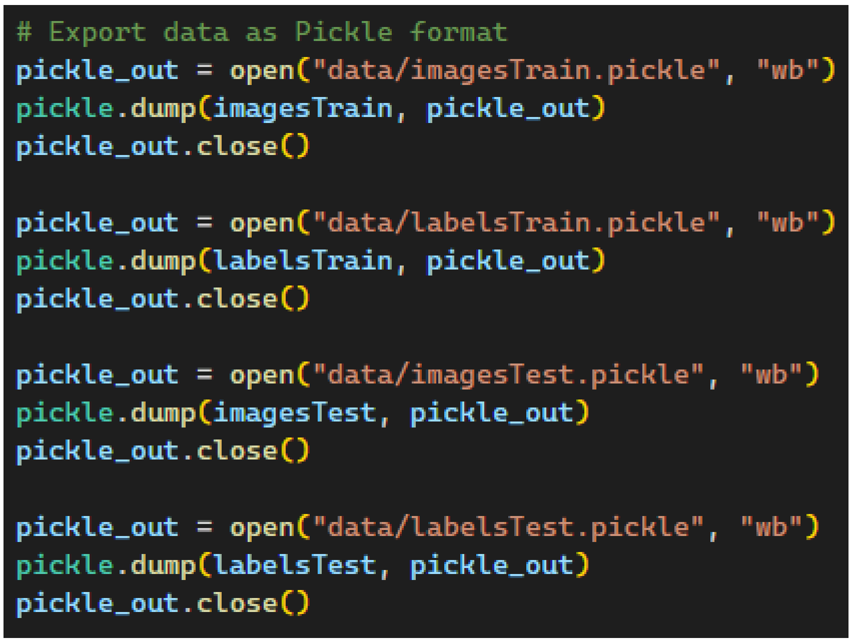
Fig. 11: One-hot formatting

Currently, the labels are categorized with values of 1 - 62, corresponding to the list of characters. The labels are in the order of:
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
(e.g. index 10 is A)

How the AI model works is that it will actually predict results for all the 62 neurons that it has. So, it will result in a range of values for all 62 neurons (ranging from 0.0 - 1.0). As a result, a conversion of the labels is necessary. Using the .to_categorical method from Keras (deep learning API which is a part of TensorFlow), we convert the labels into a one-hot format. So, instead of a value of 10 for example, the labels will return a list of 0s until the index of 10, which will return 1, followed by another list of 0s.

Graphical representation:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, …, 0]

Later on, the AI model will be able to properly see which of the 62 neurons that it predicts is actually the correct one.



```python
# Export data as Pickle format
pickle_out = open("data/imagesTrain.pickle", "wb")
pickle.dump(imagesTrain, pickle_out)
pickle_out.close()

pickle_out = open("data/labelsTrain.pickle", "wb")
pickle.dump(labelsTrain, pickle_out)
pickle_out.close()

pickle_out = open("data/imagesTest.pickle", "wb")
pickle.dump(imagesTest, pickle_out)
pickle_out.close()

pickle_out = open("data/labelsTest.pickle", "wb")
pickle.dump(labelsTest, pickle_out)
pickle_out.close()
```

Fig. 12: Converting variables to .pickle format

Lastly, using the pickle module from Python, we save these variables as its own file which we can load easily instead of needing to process the dataset every single time it needs to be accessed.

AI Model Continued

Now that our dataset is ready, we can start to train our AI model. For this project, as we are dealing with images, I will be using a convolutional neural network for the AI model. Convolutional neural networks work by dividing sections of an image as convolutions, or small regions, such as 3x3 pixels, and simplifies its values to get the most basic features of that image. Once done, it will shift over and over to eventually cover the whole image. After that, the convolutions will then be pooled, or combined, to a certain size to get its maximum value. So for example, for a 3x3 pixel region, pooling will determine a value for that 3x3 region. As we are using max pooling for this model, it will find the maximum value for that 3x3 region.
These processes are done in an attempt to get all the basic features of the image and classify a character with those set of features (edges, lines, curves, etc.).

The structure for this project's AI model:

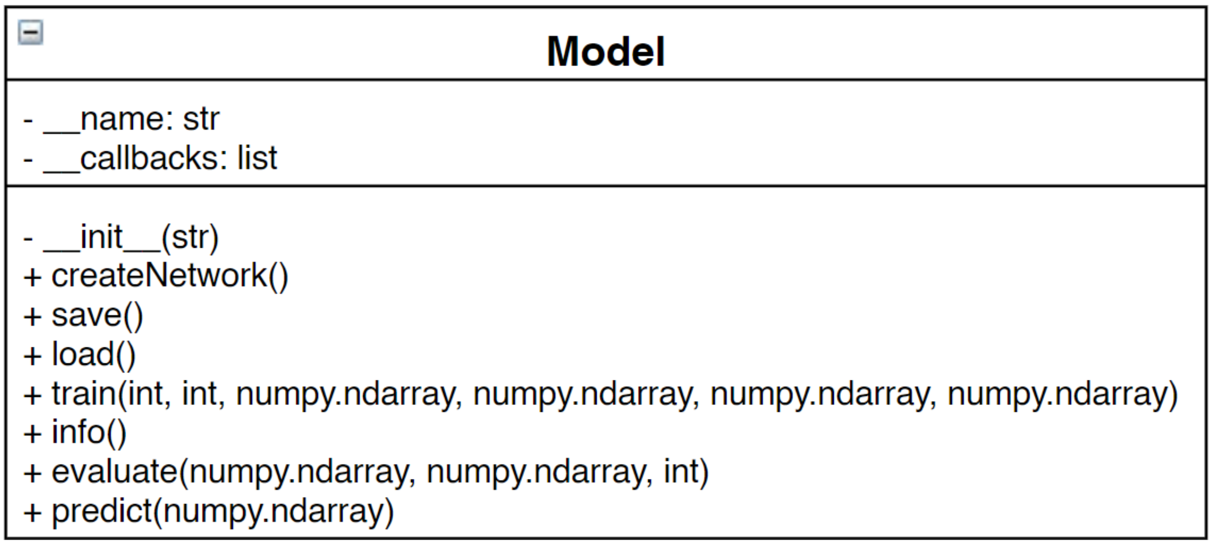
- **Input layer (Convolutional)**
- **Convolutional layer**
- **Max Pooling layer**
- **Dropout layer**
- **2x convolutional layer**
- **Max pooling layer**
- **Dropout layer**
- **Flattening layer**
- **2x hidden layer (Dense)**
- **Final dense output layer**

Dropout layers are used throughout the structure of the neural network to get rid of some of the layers that are not used throughout training. This is done to reduce what's called "overfitting", where the AI model gets very good at predicting its training dataset while not being very good at predicting what it hasn't seen (e.g. images from testing dataset).

The flattening layer is used to turn the input image (2-dimensional) back to a 1-dimensional array of values in preparation for the final output layer. After the flattening layer comes the hidden layers. Here, using dense layers, is where the image gets processed to determine how much "weight" or "value" that a neuron has towards the final prediction of the image.

The final dense output layer will produce the probabilities of all 62 neurons (values from 0.0 - 1.0) which we can then use to determine which character that the AI model predicts is most fitting from the input image by taking the highest values' index and comparing it with the one-hot formatted label.

In order to create models for testing and determining which network structure was the best, I created a Model class. Here is its UML diagram:

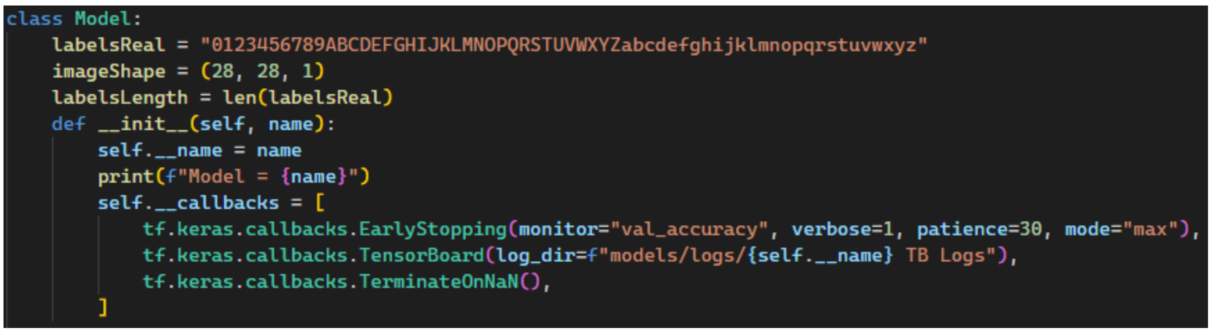


Model

- __name: str
- __callbacks: list

- __init__(str)
+ createNetwork()
+ save()
+ load()
+ train(int, int, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray)
+ info()
+ evaluate(numpy.ndarray, numpy.ndarray, int)
+ predict(numpy.ndarray)

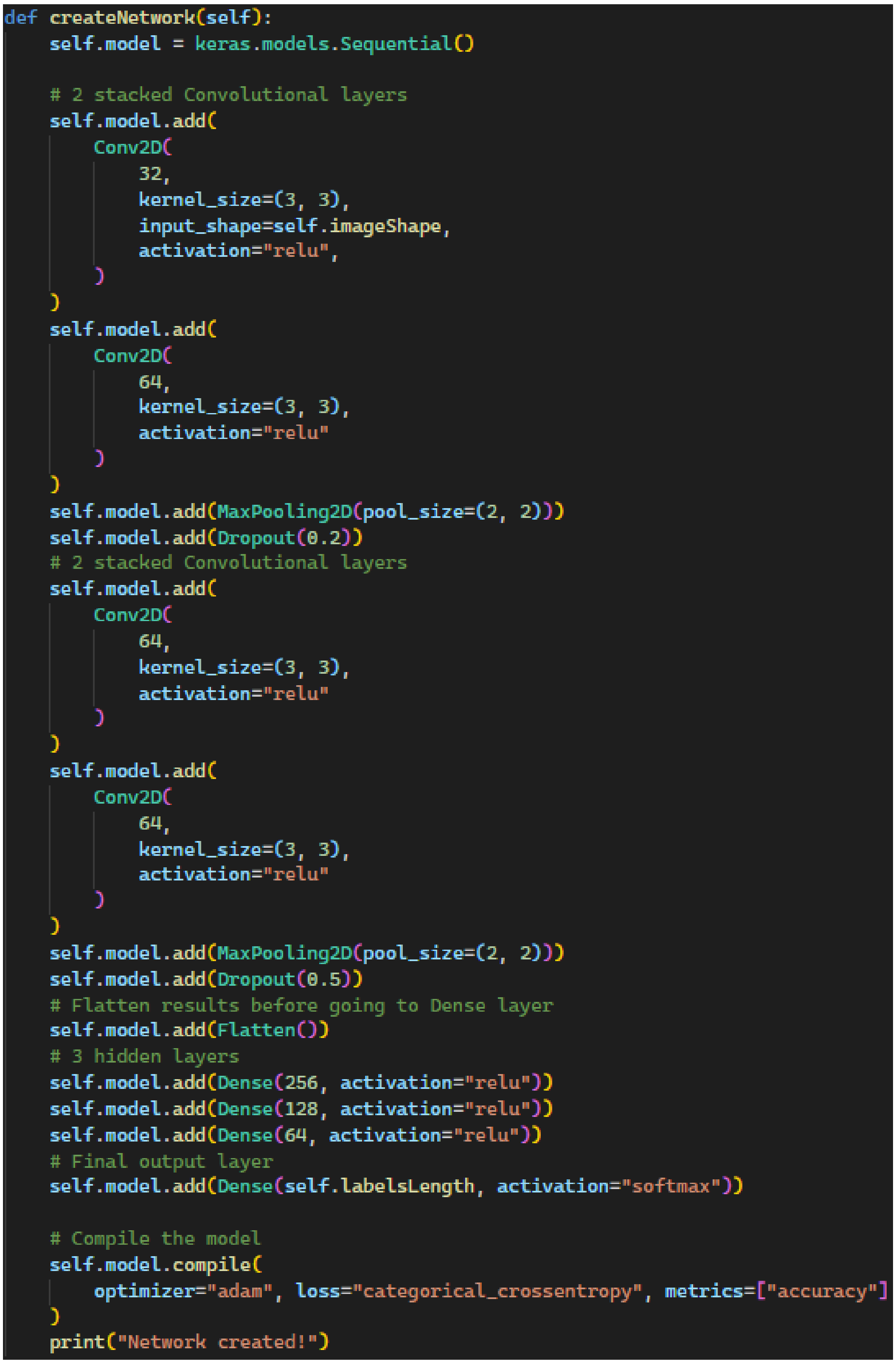
Fig. 13: Model class UML diagram



```
class Model:
    labelsReal = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
    imageShape = (28, 28, 1)
    labelsLength = len(labelsReal)
    def __init__(self, name):
        self.__name = name
        print(f"Model = {name}")
        self.__callbacks = [
            tf.keras.callbacks.EarlyStopping(monitor="val_accuracy", verbose=1, patience=30, mode="max"),
            tf.keras.callbacks.TensorBoard(log_dir=f"models/logs/{self.__name} TB Logs"),
            tf.keras.callbacks.TerminateOnNaN(),
        ]
```

Fig. 14: Model class init

The Model class is where all AI models are derived from. It holds the name of the model, as well as something called callbacks. The TensorFlow callbacks are just in there for testing purposes in order to monitor the accuracy of the model during training. The callbacks will produce a log file where we can see how the model is doing during each step of the training process. This can be viewed using TensorFlow's visualization tool kit, TensorBoard.

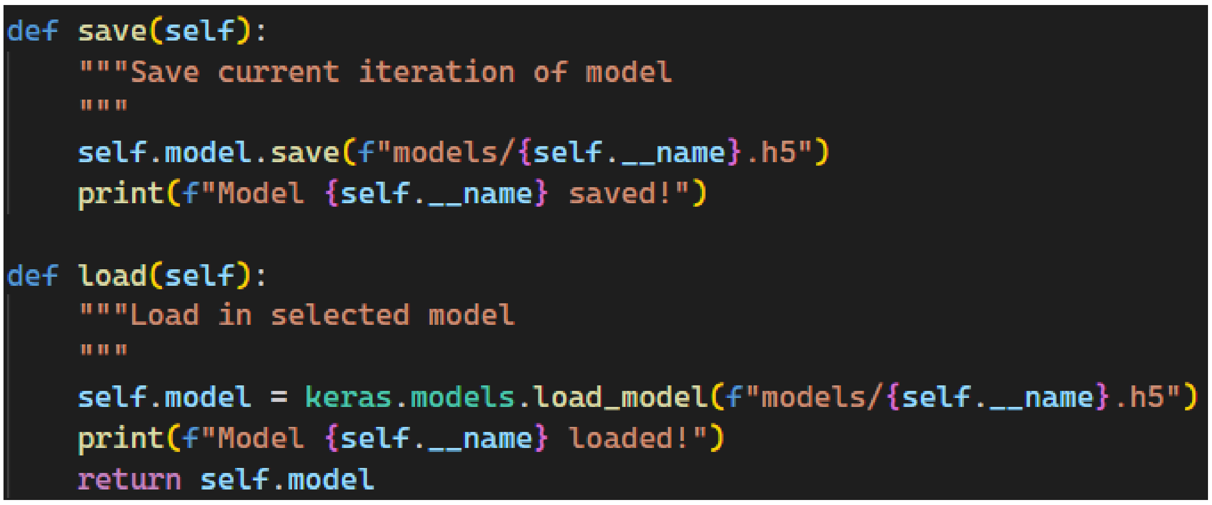


```
def createNetwork(self):
    self.model = keras.models.Sequential()

    # 2 stacked Convolutional layers
    self.model.add(
        Conv2D(
            32,
            kernel_size=(3, 3),
            input_shape=self.imageShape,
            activation="relu",
        )
    )
    self.model.add(
        Conv2D(
            64,
            kernel_size=(3, 3),
            activation="relu"
        )
    )
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.2))
    # 2 stacked Convolutional layers
    self.model.add(
        Conv2D(
            64,
            kernel_size=(3, 3),
            activation="relu"
        )
    )
    self.model.add(
        Conv2D(
            64,
            kernel_size=(3, 3),
            activation="relu"
        )
    )
    self.model.add(MaxPooling2D(pool_size=(2, 2)))
    self.model.add(Dropout(0.5))
    # Flatten results before going to Dense layer
    self.model.add(Flatten())
    # 3 hidden layers
    self.model.add(Dense(256, activation="relu"))
    self.model.add(Dense(128, activation="relu"))
    self.model.add(Dense(64, activation="relu"))
    # Final output layer
    self.model.add(Dense(self.labelsLength, activation="softmax"))

    # Compile the model
    self.model.compile(
        optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"]
    )
    print("Network created!")
```

Fig. 15: Final AI Model neural network structure

The final AI Model for this project uses these preset arguments for its neural network structure. Conv2D is for the convolutional layers, MaxPooling2D is for the pooling layers, Dropout is for the dropout layers, Flatten is for the flattening layer, Dense is for the hidden and final output layers.

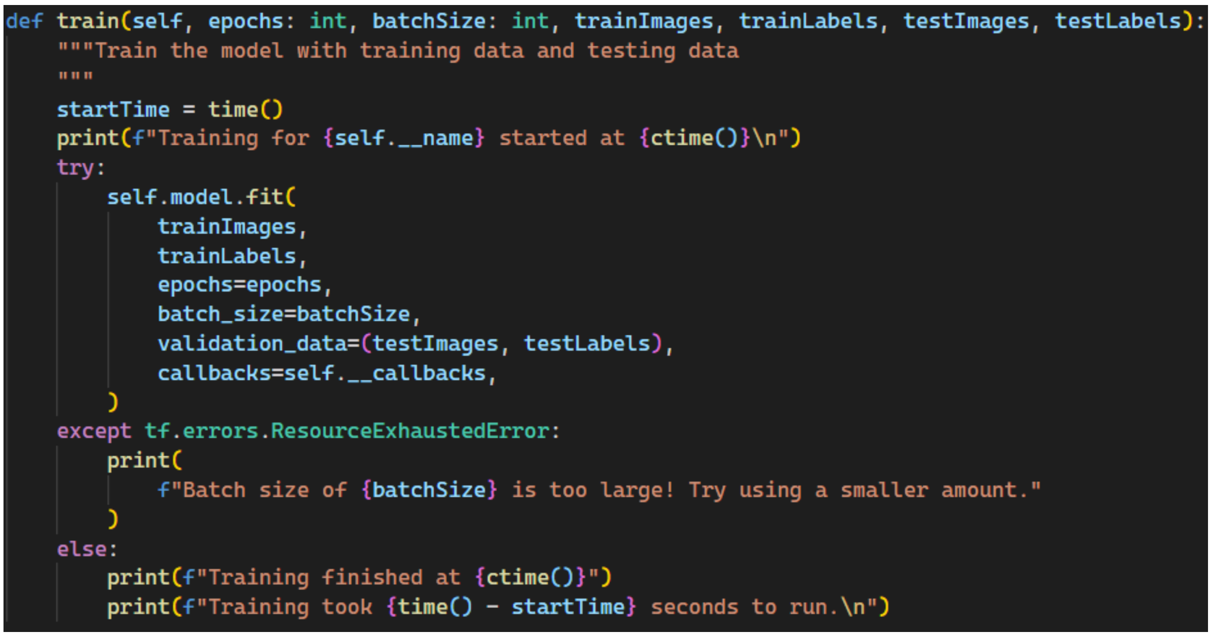
A kernel size of 3x3 is used to help speed up the process and is safer for smaller sized images (in our case, 28x28). The Relu activation function is used throughout the network in order to speed up the training process as it will output only positive values. The Softmax activation function is used at the end to calculate the probability for each node at the end. The optimizer function used is Adam, the loss function used to calculate and categorize the results is categorical cross entropy, and the metrics used to measure the success of the prediction is its accuracy.



```python
def save(self):
    """Save current iteration of model
    """
    self.model.save(f"models/{self.__name}.h5")
    print(f"Model {self.__name} saved!")

def load(self):
    """Load in selected model
    """
    self.model = keras.models.load_model(f"models/{self.__name}.h5")
    print(f"Model {self.__name} loaded!")
    return self.model
```

Fig. 16: Model save and load methods

The save and load method respectively saves and loads the current iteration of a model. The model itself is saved in one single .h5 format file for ease of use (h5py module is used to do this).

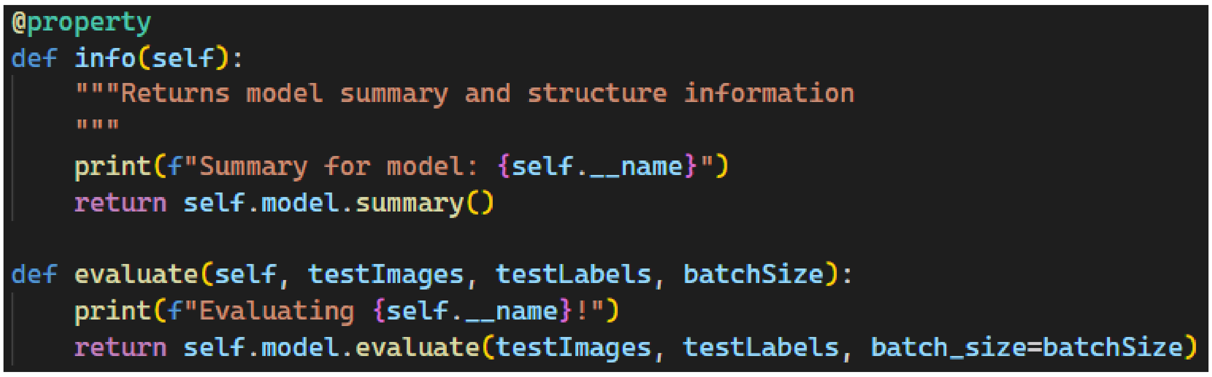


```python
def train(self, epochs: int, batchSize: int, trainImages, trainLabels, testImages, testLabels):
    """Train the model with training data and testing data
    """
    startTime = time()
    print(f"Training for {self.__name} started at {ctime()}\n")
    try:
        self.model.fit(
            trainImages,
            trainLabels,
            epochs=epochs,
            batch_size=batchSize,
            validation_data=(testImages, testLabels),
            callbacks=self.__callbacks,
        )
    except tf.errors.ResourceExhaustedError:
        print(
            f"Batch size of {batchSize} is too large! Try using a smaller amount."
        )
    else:
        print(f"Training finished at {ctime()}")
        print(f"Training took {time() - startTime} seconds to run.\n")
```

Fig. 17: Model train method

The train method takes in a few variables as its parameters. Epochs is the number of times a model wants to be trained with a certain dataset. Batch size determines the number of batches that the dataset is split up into during training. It also takes in the training images and labels for the AI model. Test images and labels are taken in for validation. This will provide data during training as to how the AI model compares against data that it is not used to train it. This helps with determining if the AI model is overfitting with its training data or not and gives an example of how the AI model would do with actual examples.

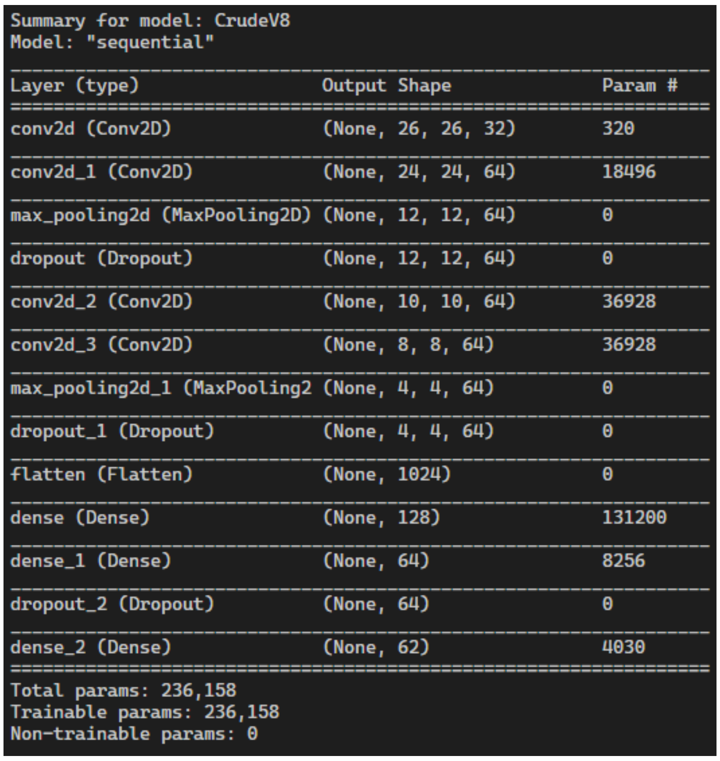
Exception handling is used here if the ResourceExhaustedError is catched as the batch size for training the model can use a large amount of system capacity. Otherwise, when the training is complete, the time module is used to determine how long it took for the model to train.



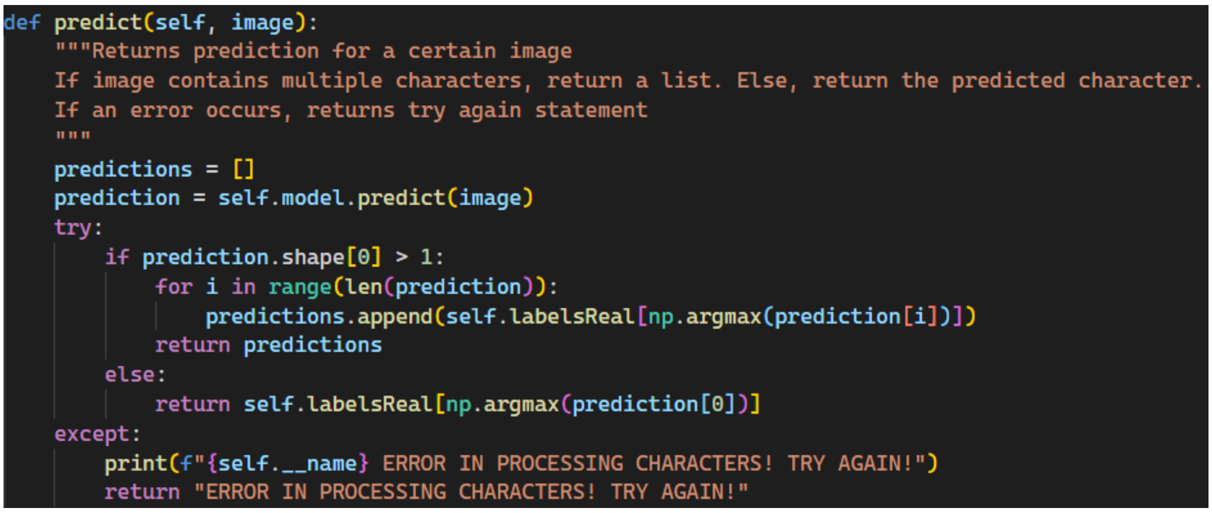
```
@property
def info(self):
    """Returns model summary and structure information
    """
    print(f"Summary for model: {self.__name}")
    return self.model.summary()

def evaluate(self, testImages, testLabels, batchSize):
    print(f"Evaluating {self.__name}!")
    return self.model.evaluate(testImages, testLabels, batch_size=batchSize)
```

Fig. 18: Model info property and evaluate method

The info property returns a summary of the model. An example of the summary:

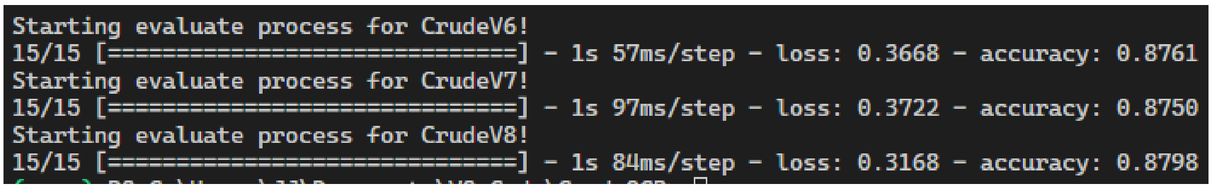

```
Summary for model: CrudeV8
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 12, 12, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 64)        36928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 4, 4, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 128)               131200
_________________________________________________________________
dense_1 (Dense)              (None, 64)                8256
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 62)                4030
=================================================================
Total params: 236,158
Trainable params: 236,158
Non-trainable params: 0
```
Fig. 19: Model example summary (CrudeV8)

Meanwhile, the evaluate method tests the trained AI Model against data and sees how accurate it is after training.


```python
def predict(self, image):
    """Returns prediction for a certain image
    If image contains multiple characters, return a list. Else, return the predicted character.
    If an error occurs, returns try again statement
    """
    predictions = []
    prediction = self.model.predict(image)
    try:
        if prediction.shape[0] > 1:
            for i in range(len(prediction)):
                predictions.append(self.labelsReal[np.argmax(prediction[i])])
            return predictions
        else:
            return self.labelsReal[np.argmax(prediction[0])]
    except:
        print(f"{self.__name} ERROR IN PROCESSING CHARACTERS! TRY AGAIN!")
        return "ERROR IN PROCESSING CHARACTERS! TRY AGAIN!"
```
Fig. 20: Model predict method

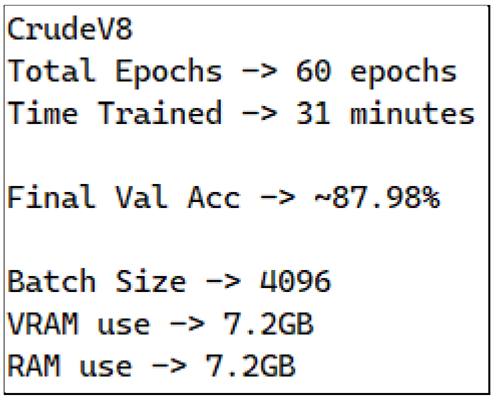
The predict method is the main method that will be used to return a prediction for an inputted image. It takes in that image and compares the prediction against self.labelsReal, which is the list of characters in order according to the one-hot formatted label. By taking the maximum value from the one-hot format, by using np.argmax, the most likely character predicted by the AI Model can be taken and returned. In the case of sudden multiple predictions, the prediction will be stored in a list first before being returned. Exception handling is used in case TensorFlow returns a random error and decides to not process the image.

And with that, an AI Model can be generated. For this project, I used a few different types of network structures before finally ending up with the current one. That's why I named the AI Model used for the project "CrudeV8".



```
Starting evaluate process for CrudeV6!
15/15 [==============================] - 1s 57ms/step - loss: 0.3668 - accuracy: 0.8761
Starting evaluate process for CrudeV7!
15/15 [==============================] - 1s 97ms/step - loss: 0.3722 - accuracy: 0.8750
Starting evaluate process for CrudeV8!
15/15 [==============================] - 1s 84ms/step - loss: 0.3168 - accuracy: 0.8798
```

Fig. 20: CrudeV8 evaluation compared to previous iterations



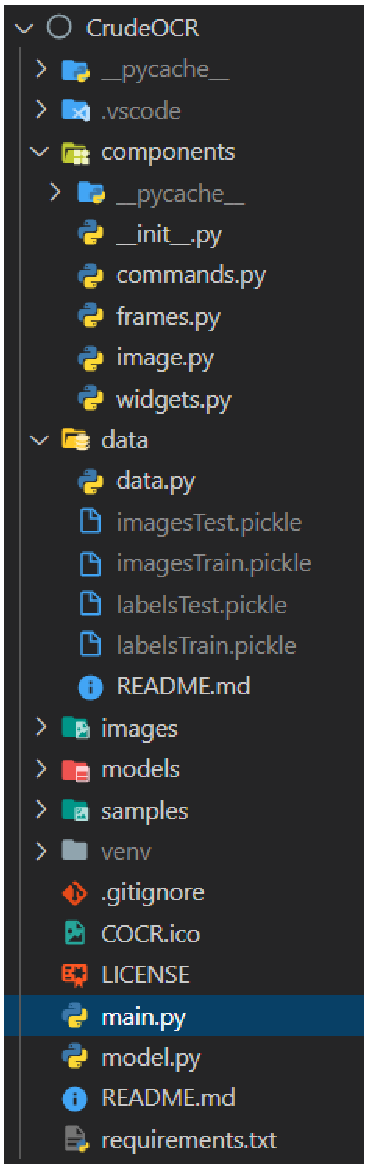
```
CrudeV8
Total Epochs -> 60 epochs
Time Trained -> 31 minutes

Final Val Acc -> ~87.98%

Batch Size -> 4096
VRAM use -> 7.2GB
RAM use -> 7.2GB
```

Fig. 21: Notes and statistics about CrudeV8 during training

And that sums up the TensorFlow part of this project.

Creating the TkInter Interface Structure

Below is the final file structure for Crude OCR:

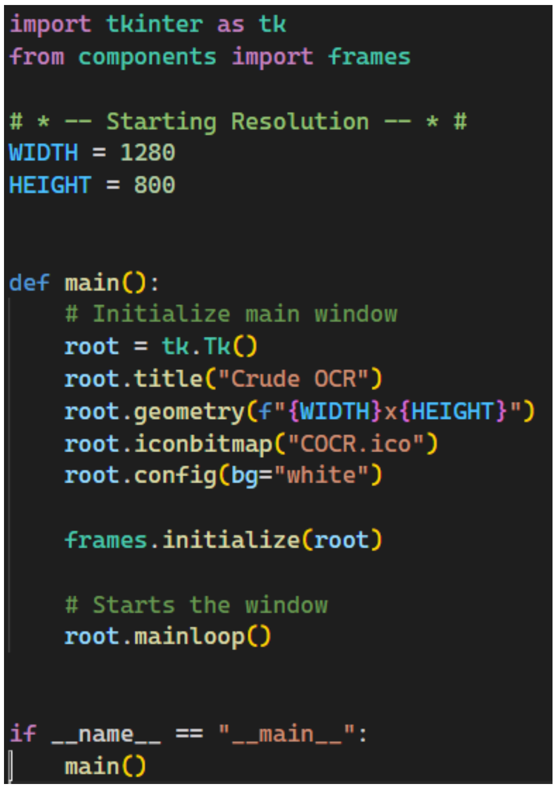


```
CrudeOCR
  __pycache__
  .vscode
  components
    __pycache__
    __init__.py
    commands.py
    frames.py
    image.py
    widgets.py
  data
    data.py
    imagesTest.pickle
    imagesTrain.pickle
    labelsTest.pickle
    labelsTrain.pickle
    README.md
  images
  models
  samples
  venv
  .gitignore
  COCR.ico
  LICENSE
  main.py
  model.py
  README.md
  requirements.txt
```

Fig. 22: Crude OCR final file structure

I decided to divide the components that make up the program into different files in a folder called components. With the __init__.py file inside the folder, Python will now treat the folder as its own package where I can import functions, classes, and etc. from.

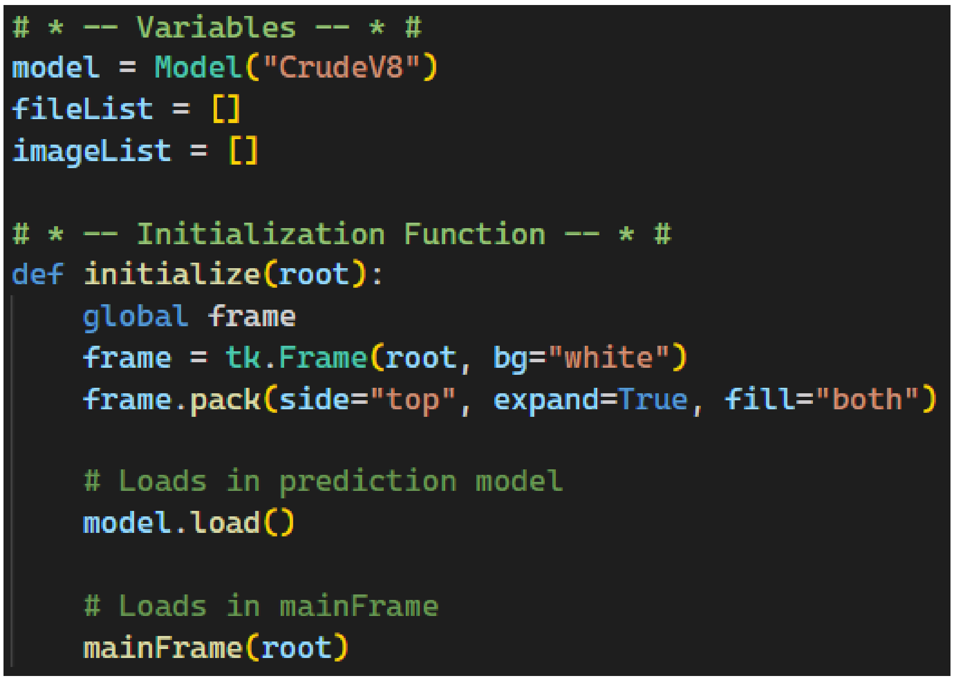
Starting with main.py:



```python
import tkinter as tk
from components import frames

# * -- Starting Resolution -- * #
WIDTH = 1280
HEIGHT = 800


def main():
    # Initialize main window
    root = tk.Tk()
    root.title("Crude OCR")
    root.geometry(f"{WIDTH}x{HEIGHT}")
    root.iconbitmap("COCR.ico")
    root.config(bg="white")

    frames.initialize(root)

    # Starts the window
    root.mainloop()


if __name__ == "__main__":
    main()
```

Fig. 23: main.py file contents

main.py consists of the very basics of getting a TkInter window initialized. The root variable is initialized as a tkinter.Tk class. We then configure its title, window size, icon (visible in Windows only sadly), and background. Users can configure the starting resolution before starting the window, though the program itself still supports dynamic resizing even after it's started. Before starting the window with the .mainloop method, we import the frames file from the components package to get the basic TkInter widgets up. All of this is wrapped in a main() function and will only run if it is in its original folder (if __name__ == "__main__"). Running this file will start everything up but before that, a small explanation for each frame is needed.



```python
# * -- Variables -- * #
model = Model("CrudeV8")
fileList = []
imageList = []

# * -- Initialization Function -- * #
def initialize(root):
    global frame
    frame = tk.Frame(root, bg="white")
    frame.pack(side="top", expand=True, fill="both")

    # Loads in prediction model
    model.load()

    # Loads in mainFrame
    mainFrame(root)
```

Fig. 24: initialize function inside frames.py

The frames.initialize function in main.py refers to this function. The AI Model is loaded when the file is referenced along with 2 lists, fileList and imageList, which will be explained later. The initialize function takes in the root tk.Tk object from main.py which is then passed on to frame. Frame is where all the widgets for the project will be placed in. The initialize function also loads in the main menu frame.

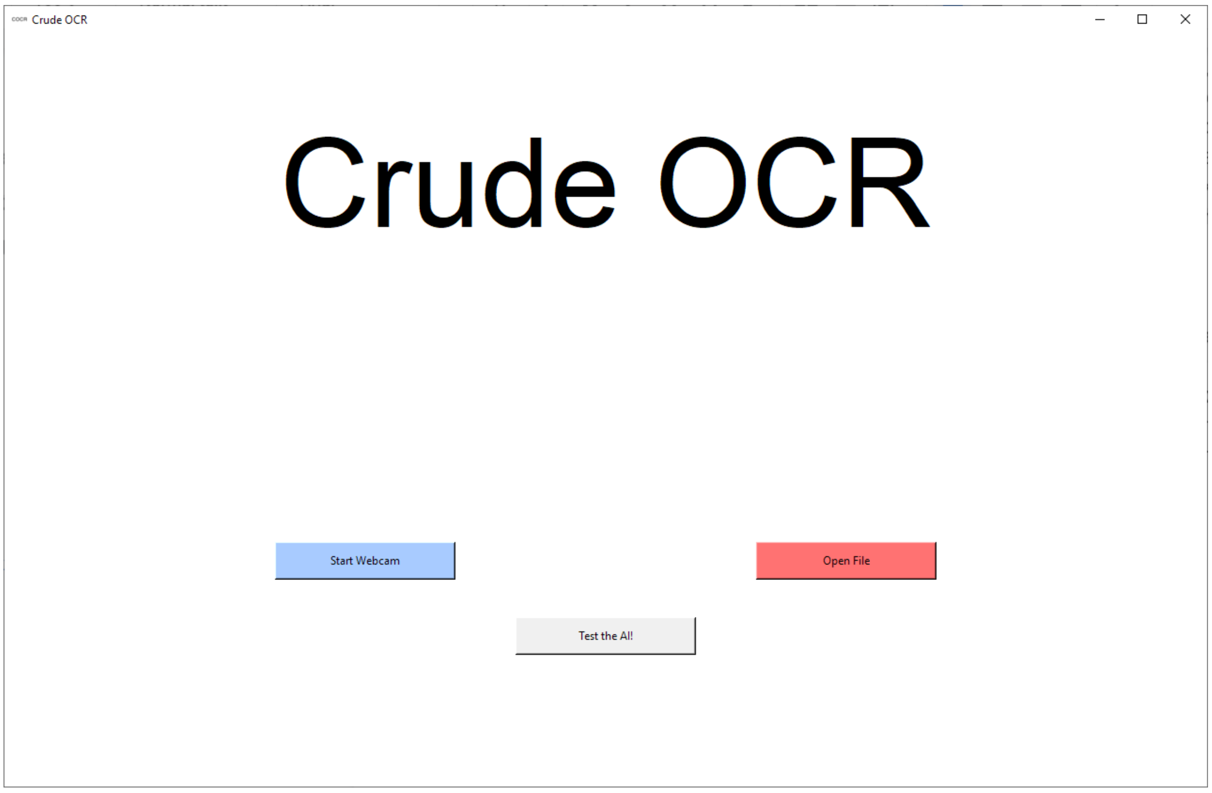
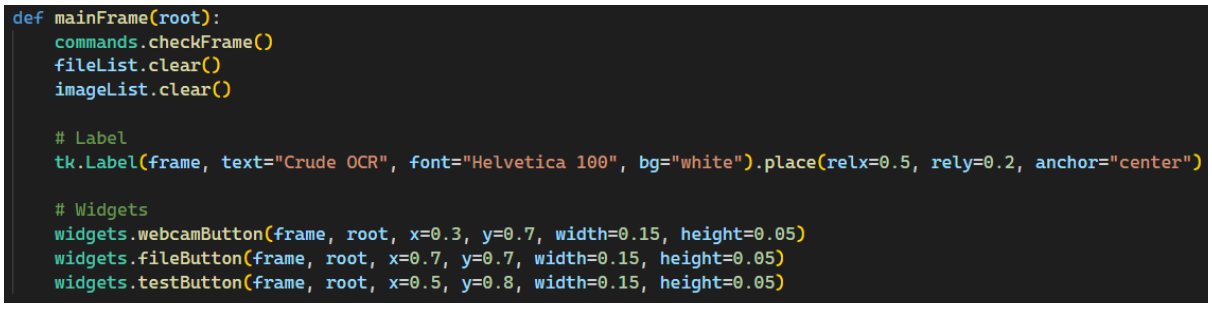


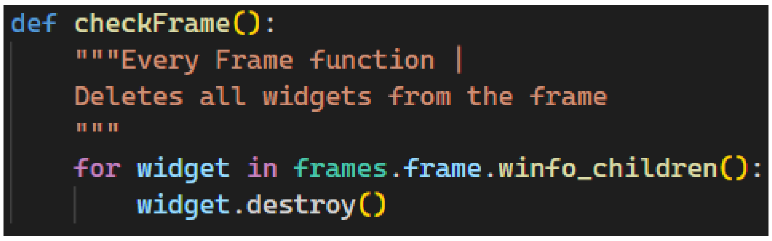
Fig. 25: Crude OCR mainFrame



```python
def mainFrame(root):
    commands.checkFrame()
    fileList.clear()
    imageList.clear()

    # Label
    tk.Label(frame, text="Crude OCR", font="Helvetica 100", bg="white").place(relx=0.5, rely=0.2, anchor="center")

    # Widgets
    widgets.webcamButton(frame, root, x=0.3, y=0.7, width=0.15, height=0.05)
    widgets.fileButton(frame, root, x=0.7, y=0.7, width=0.15, height=0.05)
    widgets.testButton(frame, root, x=0.5, y=0.8, width=0.15, height=0.05)
```

Fig. 26: mainFrame function

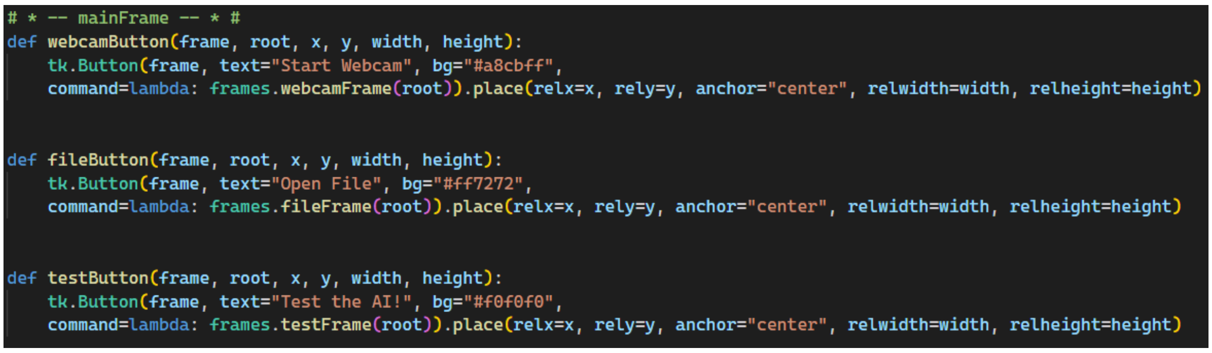
The mainFrame is the hub for the program. The back button in all the other frames always leads to this frame. When displayed, it will clear the fileList and imageList (will be explained later) and also calls the checkFrame command from the commands file.



```python
def checkFrame():
    """Every Frame function |
    Deletes all widgets from the frame
    """
    for widget in frames.frame.winfo_children():
        widget.destroy()
```

Fig. 27: checkFrame function

The checkFrame function is placed in every frames' start and is used to clear all the widgets from the previous frame. Back to the mainFrame explanation. Instead of instantiating new variables for the widgets, I structured it so that prewritten functions from the widgets file create and place the widgets according to the arguments that are passed in. They also take in other arguments needed if the widget's command needs an argument to be passed in.



```python
# * -- mainFrame -- * #
def webcamButton(frame, root, x, y, width, height):
    tk.Button(frame, text="Start Webcam", bg="#a8cbff",
    command=lambda: frames.webcamFrame(root)).place(relx=x, rely=y, anchor="center", relwidth=width, relheight=height)


def fileButton(frame, root, x, y, width, height):
    tk.Button(frame, text="Open File", bg="#ff7272",
    command=lambda: frames.fileFrame(root)).place(relx=x, rely=y, anchor="center", relwidth=width, relheight=height)


def testButton(frame, root, x, y, width, height):
    tk.Button(frame, text="Test the AI!", bg="#f0f0f0",
    command=lambda: frames.testFrame(root)).place(relx=x, rely=y, anchor="center", relwidth=width, relheight=height)
```

Fig. 28: mainFrame widgets

Webcam Frame

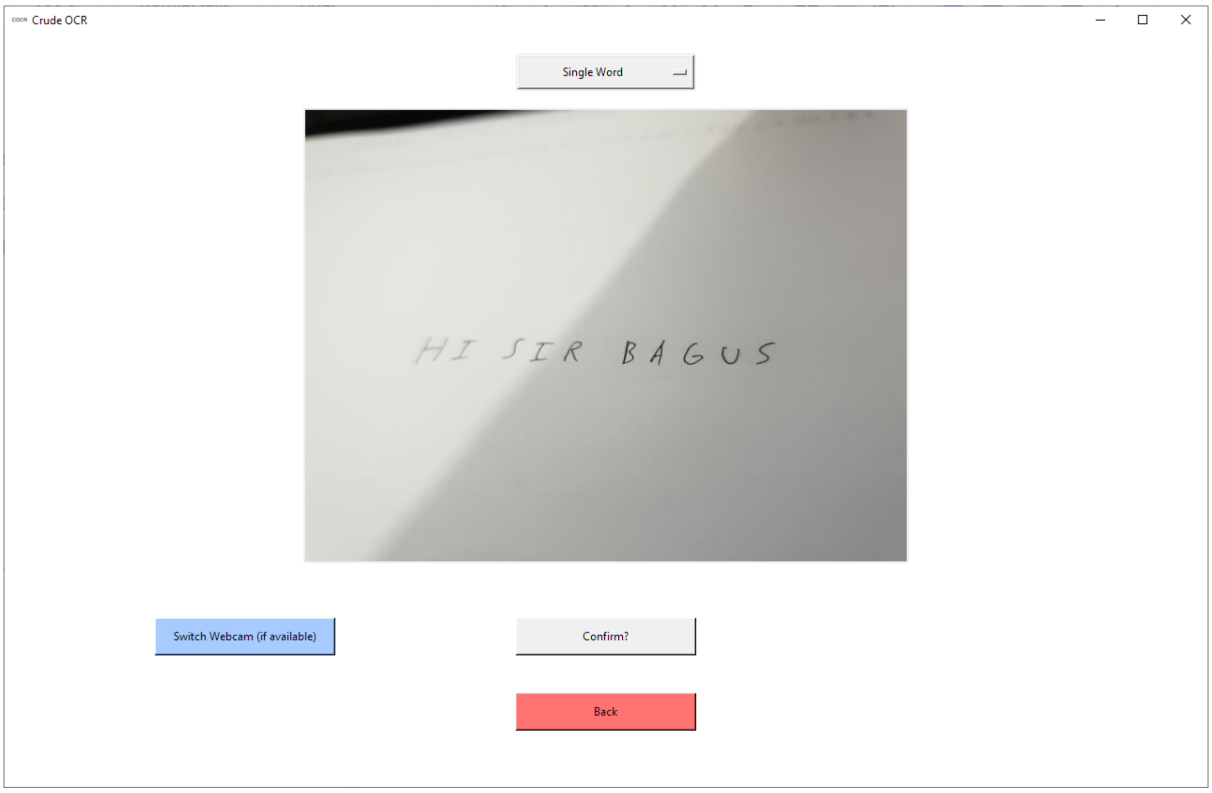
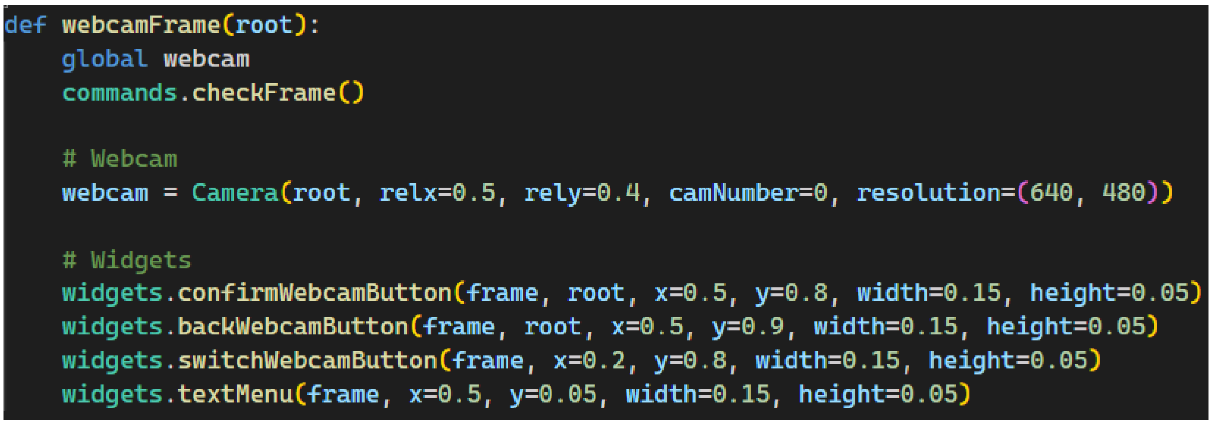


Fig. 28: Crude OCR webcamFrame



```
def webcamFrame(root):
    global webcam
    commands.checkFrame()

    # Webcam
    webcam = Camera(root, relx=0.5, rely=0.4, camNumber=0, resolution=(640, 480))

    # Widgets
    widgets.confirmWebcamButton(frame, root, x=0.5, y=0.8, width=0.15, height=0.05)
    widgets.backWebcamButton(frame, root, x=0.5, y=0.9, width=0.15, height=0.05)
    widgets.switchWebcamButton(frame, x=0.2, y=0.8, width=0.15, height=0.05)
    widgets.textMenu(frame, x=0.5, y=0.05, width=0.15, height=0.05)
```

Fig. 29: webcamFrame function

The webcamFrame initializes with an object of the class Camera. Here is its UML diagram:

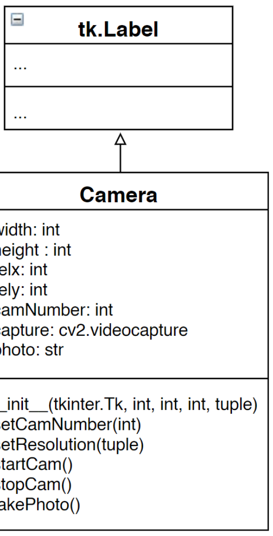
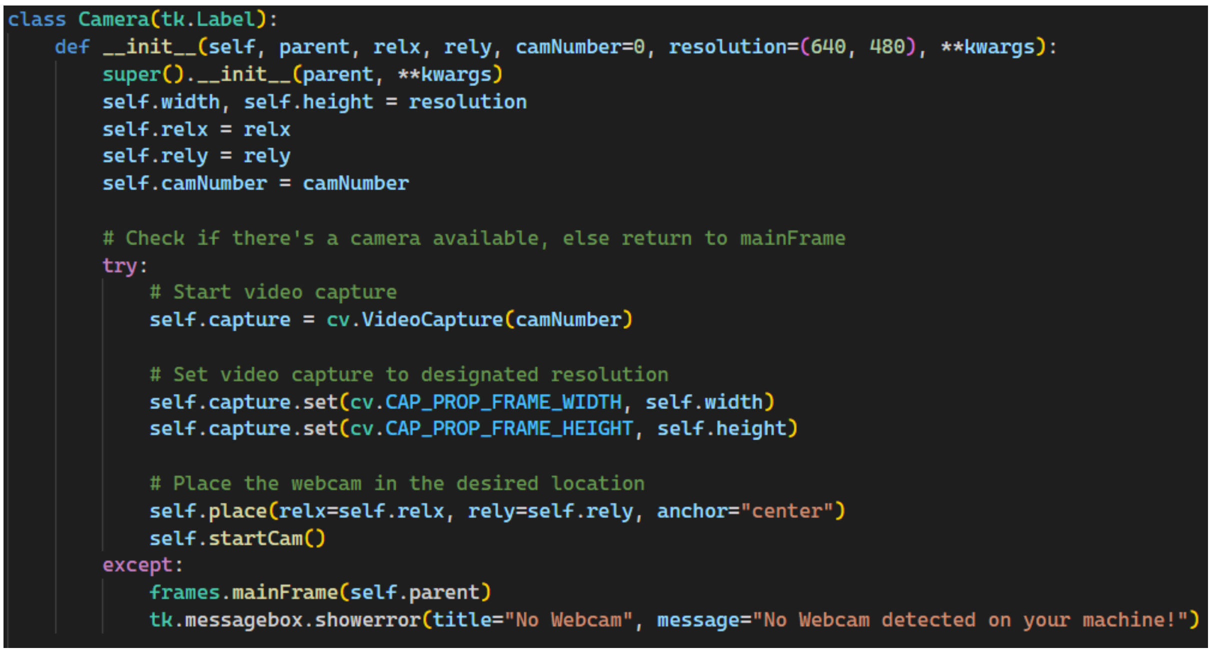


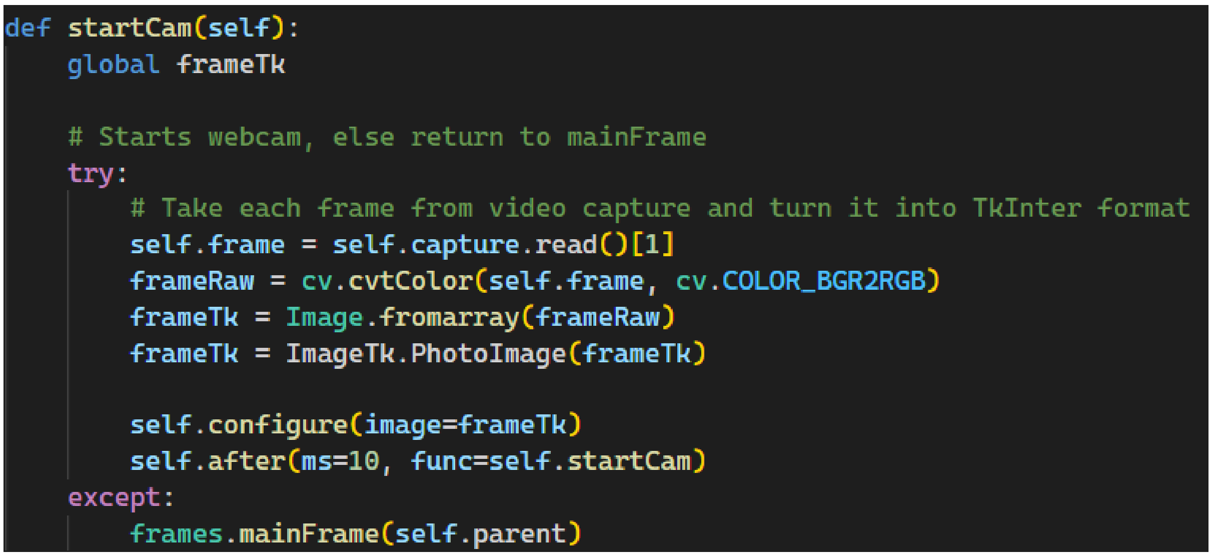
Fig. 30: Camera class UML diagram



```python
class Camera(tk.Label):
    def __init__(self, parent, relx, rely, camNumber=0, resolution=(640, 480), **kwargs):
        super().__init__(parent, **kwargs)
        self.width, self.height = resolution
        self.relx = relx
        self.rely = rely
        self.camNumber = camNumber

        # Check if there's a camera available, else return to mainFrame
        try:
            # Start video capture
            self.capture = cv.VideoCapture(camNumber)

            # Set video capture to designated resolution
            self.capture.set(cv.CAP_PROP_FRAME_WIDTH, self.width)
            self.capture.set(cv.CAP_PROP_FRAME_HEIGHT, self.height)

            # Place the webcam in the desired location
            self.place(relx=self.relx, rely=self.rely, anchor="center")
            self.startCam()
        except:
            frames.mainFrame(self.parent)
            tk.messagebox.showerror(title="No Webcam", message="No Webcam detected on your machine!")
```

Fig. 31: Camera class init

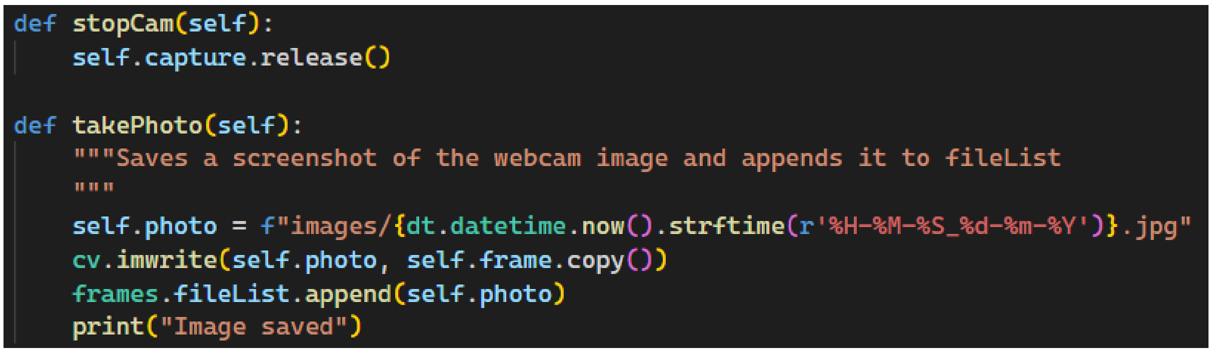
The Camera class inherits the functionality of the tk.Label class for convenience's sake. The camNumber parameter determines which camera wants to be opened by the user. For users that don’t have a webcam, the exception handling takes care of it and returns the user back to the mainFrame while displaying a pop up error message box. Otherwise, self.width and self.height determines the camera’s displayed resolution and the startCam method is called.



```python
def startCam(self):
    global frameTk

    # Starts webcam, else return to mainFrame
    try:
        # Take each frame from video capture and turn it into TkInter format
        self.frame = self.capture.read()[1]
        frameRaw = cv.cvtColor(self.frame, cv.COLOR_BGR2RGB)
        frameTk = Image.fromarray(frameRaw)
        frameTk = ImageTk.PhotoImage(frameTk)

        self.configure(image=frameTk)
        self.after(ms=10, func=self.startCam)
    except:
        frames.mainFrame(self.parent)
```

Fig. 32: Camera startCam method

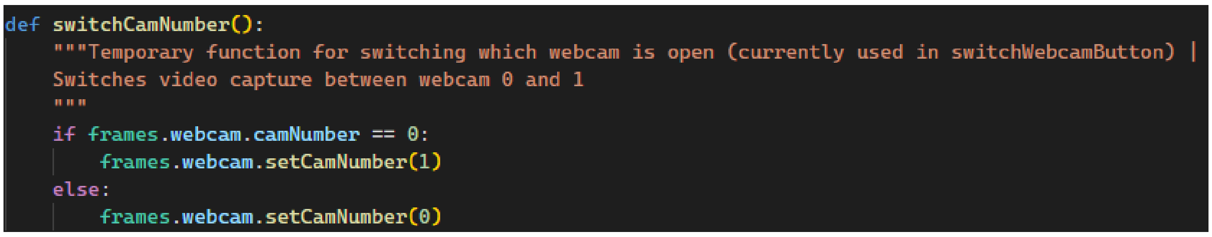
The startCam method takes frames from the webcam and converts it into a format that TkInter can read. This utilizes the Pillows module's class methods fromarray and PhotoImage. After 10 milliseconds, the function will run recursively to keep updating the webcam displayed with the newest frames.



```python
def stopCam(self):
    self.capture.release()

def takePhoto(self):
    """Saves a screenshot of the webcam image and appends it to fileList
    """
    self.photo = f"images/{dt.datetime.now().strftime(r'%H-%M-%S_%d-%m-%Y')}.jpg"
    cv.imwrite(self.photo, self.frame.copy())
    frames.fileList.append(self.photo)
    print("Image saved")
```

Fig. 33: Camera stopCam method

The stopCam and takePhoto methods are used when the "Confirm?" button in the webcamFrame is clicked. It stops the webcam's capture feed, and takes a photo that's saved in the images folder inside the program. It also appends the photo taken into fileList (explanation is still later on).



```python
def switchCamNumber():
    """Temporary function for switching which webcam is open (currently used in switchWebcamButton) |
    Switches video capture between webcam 0 and 1
    """
    if frames.webcam.camNumber == 0:
        frames.webcam.setCamNumber(1)
    else:
        frames.webcam.setCamNumber(0)
```

Fig. 34: switchCamNumber function

There is currently no method that can be used to list all the available cameras that are available on the user's machine, so a temporary switchCamNumber function is connected to the switchWebcamButton in webcamFrame.

File Frame

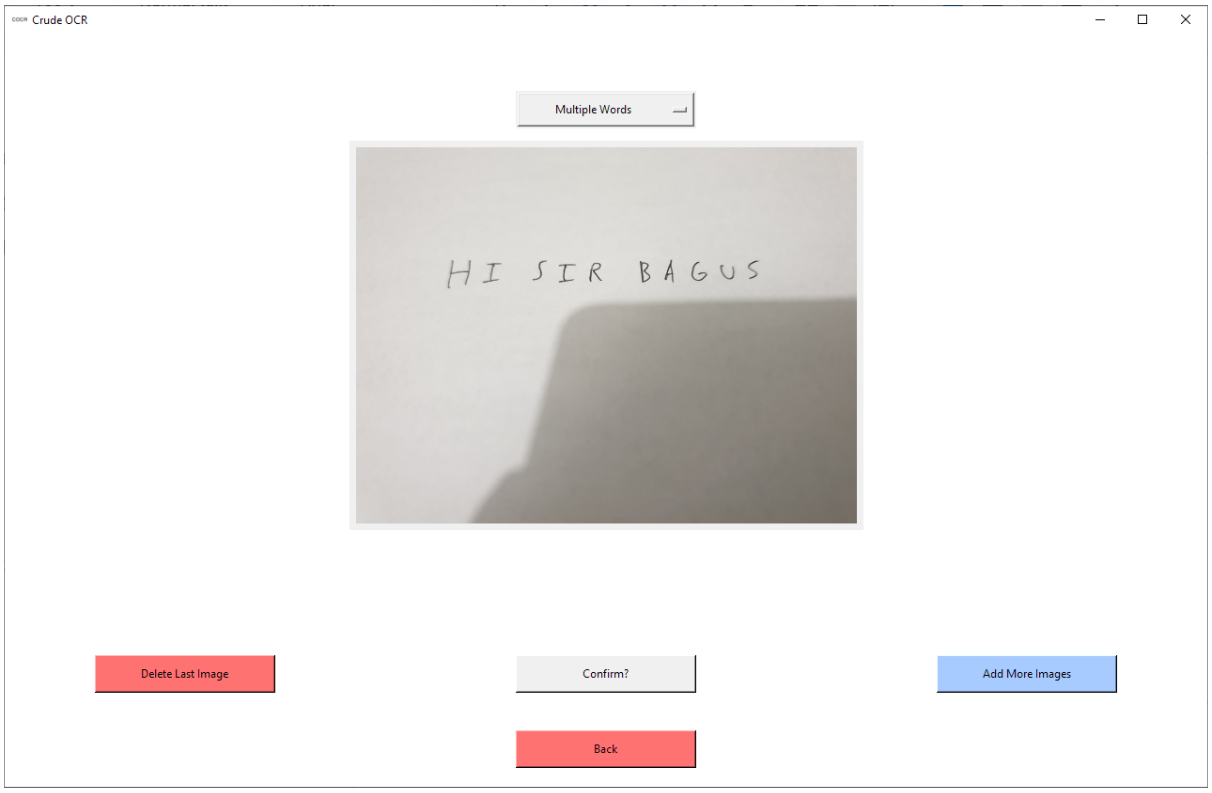
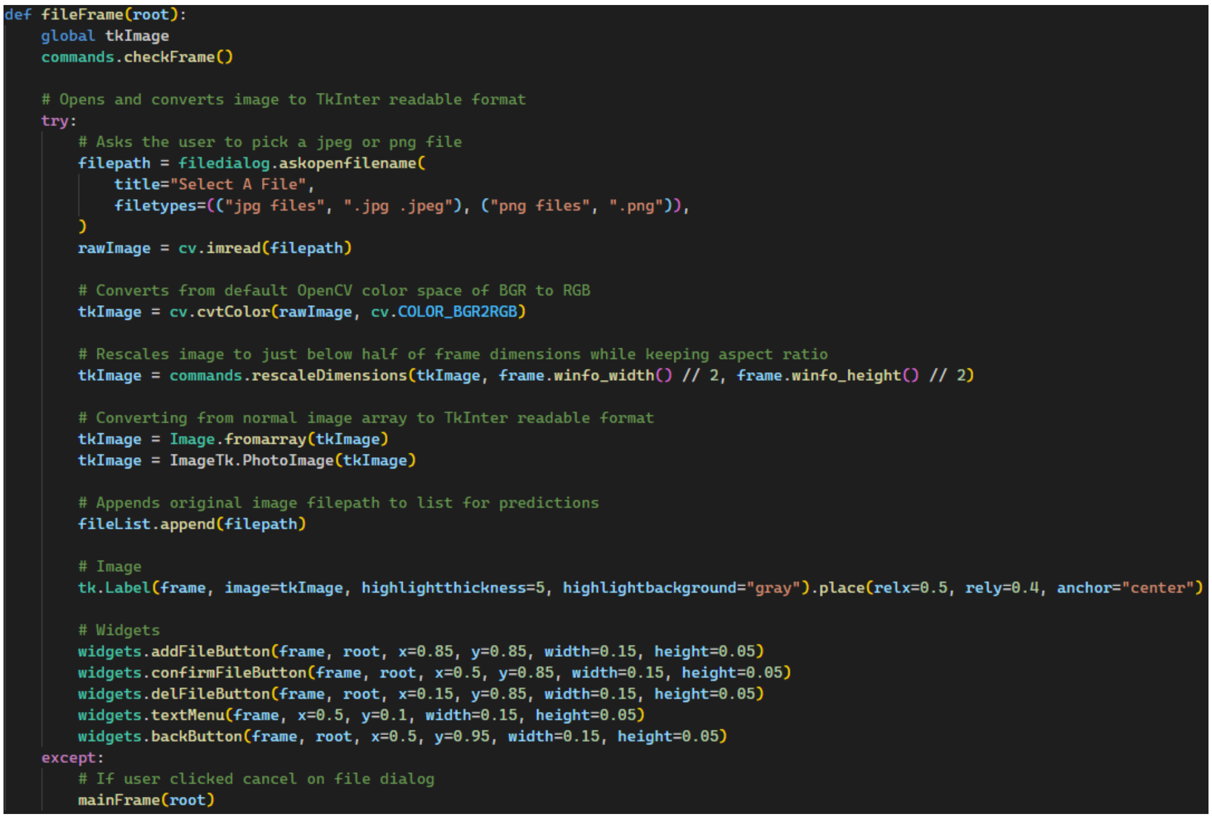


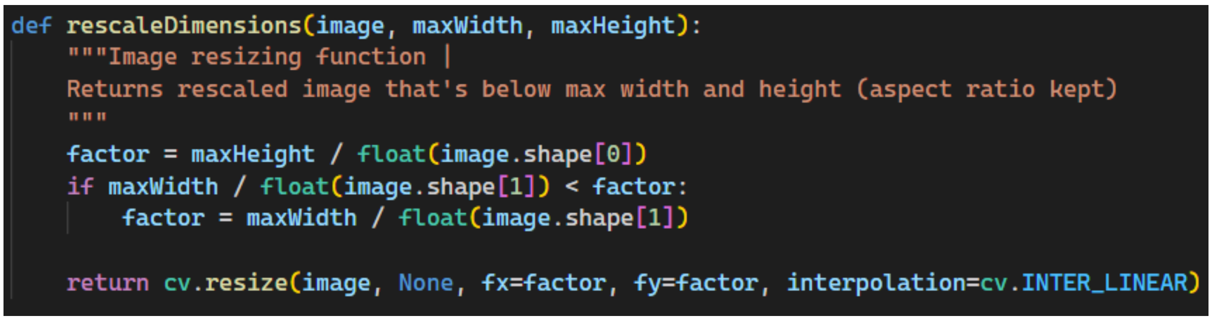
Fig. 35: Crude OCR fileFrame



```python
def fileFrame(root):
    global tkImage
    commands.checkFrame()

    # Opens and converts image to TkInter readable format
    try:
        # Asks the user to pick a jpeg or png file
        filepath = filedialog.askopenfilename(
            title="Select A File",
            filetypes=(("jpg files", ".jpg .jpeg"), ("png files", ".png")),
        )
        rawImage = cv.imread(filepath)

        # Converts from default OpenCV color space of BGR to RGB
        tkImage = cv.cvtColor(rawImage, cv.COLOR_BGR2RGB)

        # Rescales image to just below half of frame dimensions while keeping aspect ratio
        tkImage = commands.rescaleDimensions(tkImage, frame.winfo_width() // 2, frame.winfo_height() // 2)

        # Converting from normal image array to TkInter readable format
        tkImage = Image.fromarray(tkImage)
        tkImage = ImageTk.PhotoImage(tkImage)

        # Appends original image filepath to list for predictions
        fileList.append(filepath)

        # Image
        tk.Label(frame, image=tkImage, highlightthickness=5, highlightbackground="gray").place(relx=0.5, rely=0.4, anchor="center")

        # Widgets
        widgets.addFileButton(frame, root, x=0.85, y=0.85, width=0.15, height=0.05)
        widgets.confirmFileButton(frame, root, x=0.5, y=0.85, width=0.15, height=0.05)
        widgets.delFileButton(frame, root, x=0.15, y=0.85, width=0.15, height=0.05)
        widgets.textMenu(frame, x=0.5, y=0.1, width=0.15, height=0.05)
        widgets.backButton(frame, root, x=0.5, y=0.95, width=0.15, height=0.05)
    except:
        # If user clicked cancel on file dialog
        mainFrame(root)
```

Fig. 36: fileFrame function

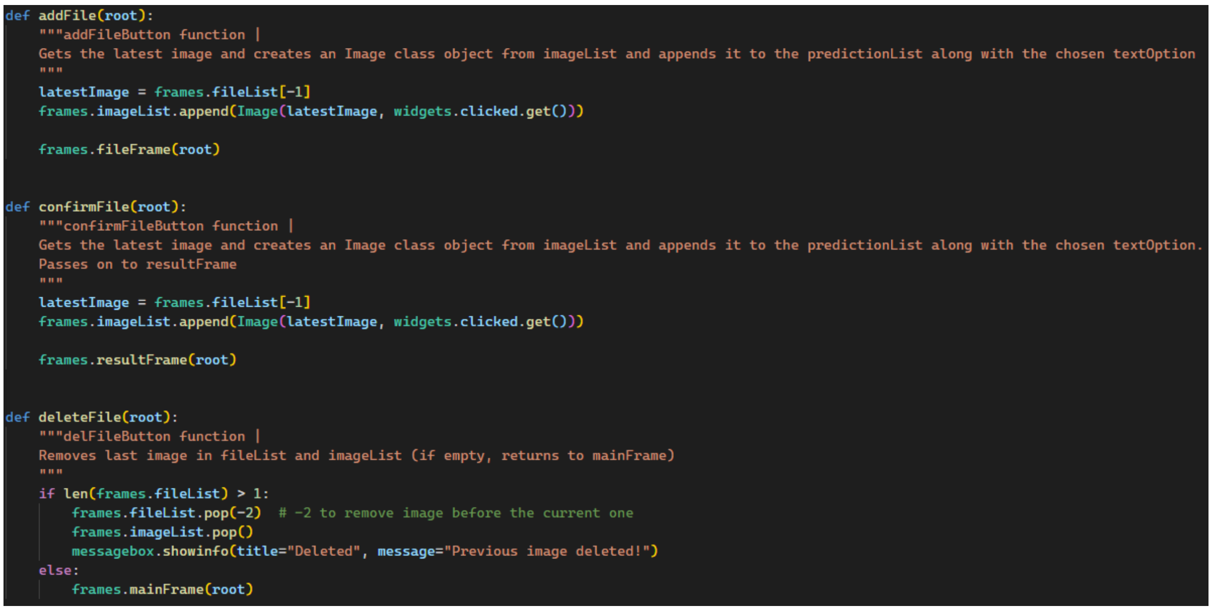
In addition to the normal widgets, fileFrame contains the logic to how the chosen image will be displayed and handled for the user to view. (Explanation for this is in the comments)



```python
def rescaleDimensions(image, maxWidth, maxHeight):
    """Image resizing function |
    Returns rescaled image that's below max width and height (aspect ratio kept)
    """
    factor = maxHeight / float(image.shape[0])
    if maxWidth / float(image.shape[1]) < factor:
        factor = maxWidth / float(image.shape[1])

    return cv.resize(image, None, fx=factor, fy=factor, interpolation=cv.INTER_LINEAR)
```

Fig. 37: rescaleDimensions function

The rescaleDimensions function is called when processing the chosen image. It takes in the maximum width and height that an image can be and resizes it while keeping its aspect ratio. This function is called again later on in the program, so keep it in mind. In the case of fileFrame, the image will be resized according to the program window's current width and height in half so that it doesn't obstruct the other widgets.



```python
def addFile(root):
    """addFileButton function |
    Gets the latest image and creates an Image class object from imageList and appends it to the predictionList along with the chosen textOption
    """
    latestImage = frames.fileList[-1]
    frames.imageList.append(Image(latestImage, widgets.clicked.get()))

    frames.fileFrame(root)


def confirmFile(root):
    """confirmFileButton function |
    Gets the latest image and creates an Image class object from imageList and appends it to the predictionList along with the chosen textOption.
    Passes on to resultFrame
    """
    latestImage = frames.fileList[-1]
    frames.imageList.append(Image(latestImage, widgets.clicked.get()))

    frames.resultFrame(root)


def deleteFile(root):
    """delFileButton function |
    Removes last image in fileList and imageList (if empty, returns to mainFrame)
    """
    if len(frames.fileList) > 1:
        frames.fileList.pop(-2)  # -2 to remove image before the current one
        frames.imageList.pop()
        messagebox.showinfo(title="Deleted", message="Previous image deleted!")
    else:
        frames.mainFrame(root)
```

Fig. 38: addFile, confirmFile, and deleteFile functions

The confirm, add, and delete buttons on the fileFrame calls on these functions from the commands file.
The addFileButton takes the latest image appended in fileList (the currently viewed one), and appends it as an Image class object in imageList along with the current text option chosen from the option menu. It will then return to fileFrame where the process can be repeated however many times the user wants. This is also the same with the confirmFileButton, except that it leads the user straight to resultFrame instead of returning back to resultFrame again. The delFileButton removes the previous image from the fileList if there is one, otherwise it will lead the user back to mainFrame. (Image class will be explained later)

Testing Frame

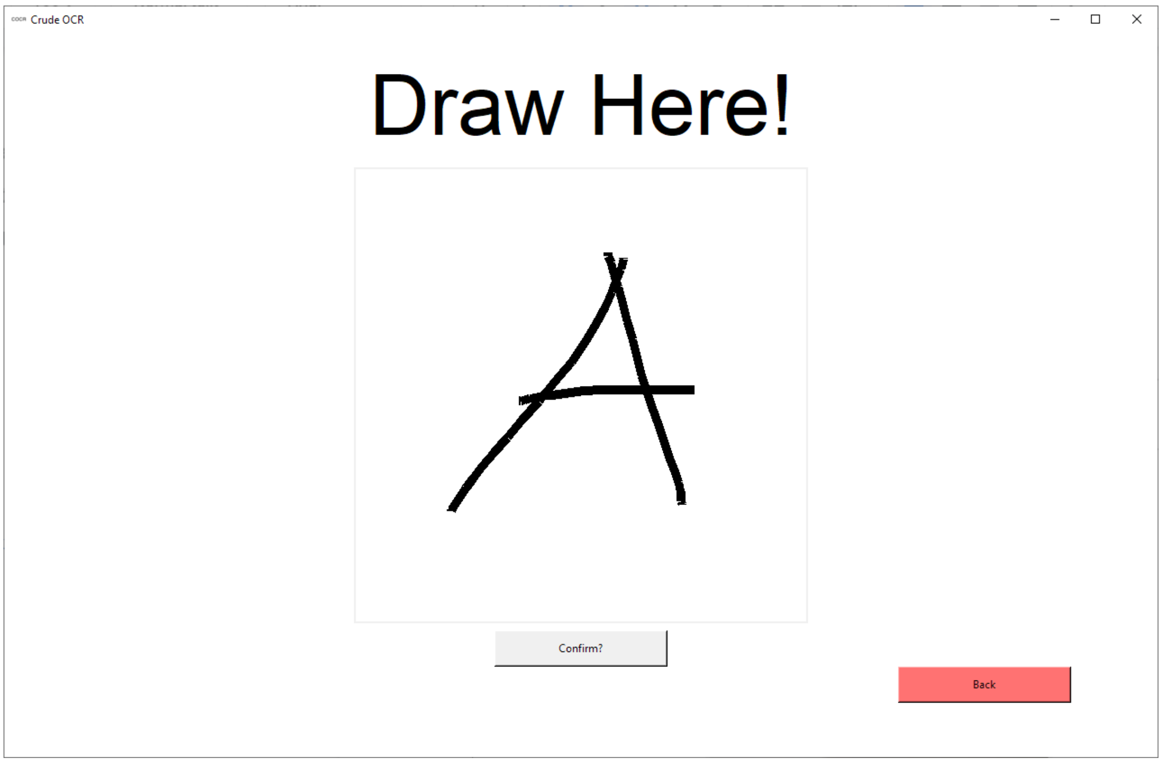
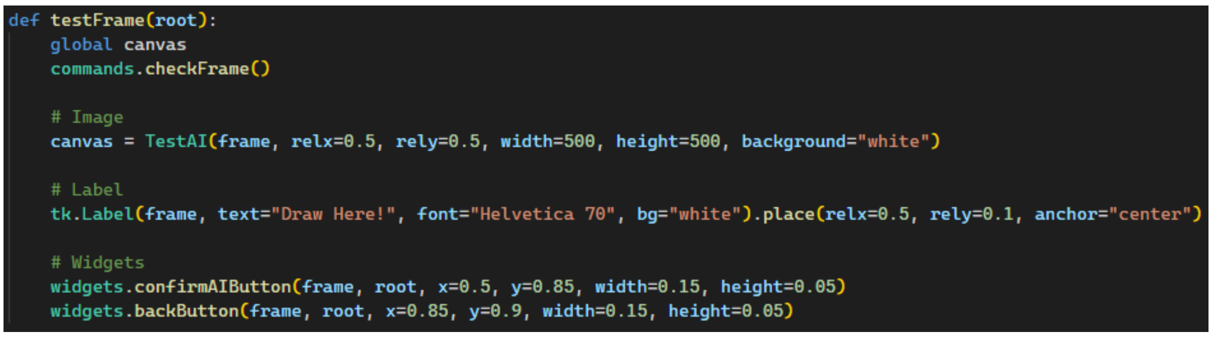


Fig. 39: Crude OCR testFrame



```
def testFrame(root):
    global canvas
    commands.checkFrame()

    # Image
    canvas = TestAI(frame, relx=0.5, rely=0.5, width=500, height=500, background="white")

    # Label
    tk.Label(frame, text="Draw Here!", font="Helvetica 70", bg="white").place(relx=0.5, rely=0.1, anchor="center")

    # Widgets
    widgets.confirmAIButton(frame, root, x=0.5, y=0.85, width=0.15, height=0.05)
    widgets.backButton(frame, root, x=0.85, y=0.9, width=0.15, height=0.05)
```

Fig. 40: testFrame function

The testFrame initializes with an object of the class TestAI. Here is its UML diagram:

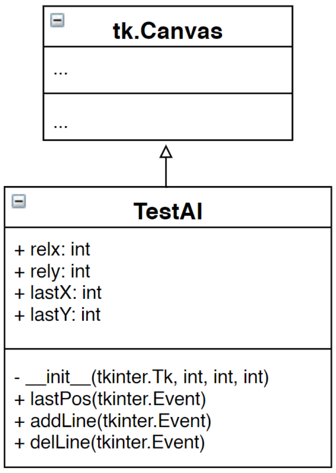
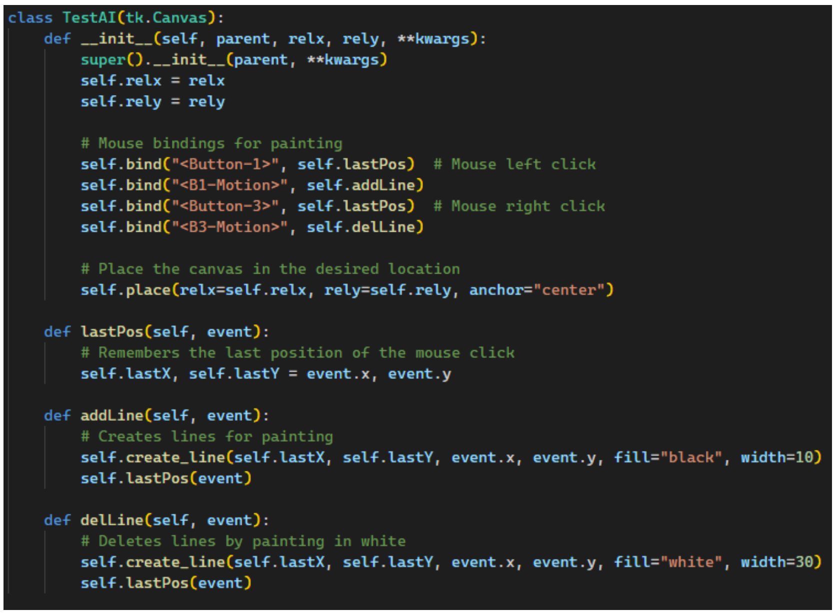


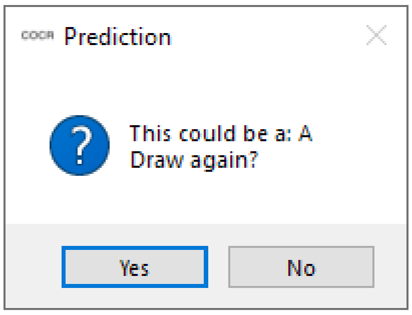
Fig. 41: TestAI class UML diagram



```python
class TestAI(tk.Canvas):
    def __init__(self, parent, relx, rely, **kwargs):
        super().__init__(parent, **kwargs)
        self.relx = relx
        self.rely = rely

        # Mouse bindings for painting
        self.bind("<Button-1>", self.lastPos)  # Mouse left click
        self.bind("<B1-Motion>", self.addLine)
        self.bind("<Button-3>", self.lastPos)  # Mouse right click
        self.bind("<B3-Motion>", self.delLine)

        # Place the canvas in the desired location
        self.place(relx=self.relx, rely=self.rely, anchor="center")

    def lastPos(self, event):
        # Remembers the last position of the mouse click
        self.lastX, self.lastY = event.x, event.y

    def addLine(self, event):
        # Creates lines for painting
        self.create_line(self.lastX, self.lastY, event.x, event.y, fill="black", width=10)
        self.lastPos(event)

    def delLine(self, event):
        # Deletes lines by painting in white
        self.create_line(self.lastX, self.lastY, event.x, event.y, fill="white", width=30)
        self.lastPos(event)
```

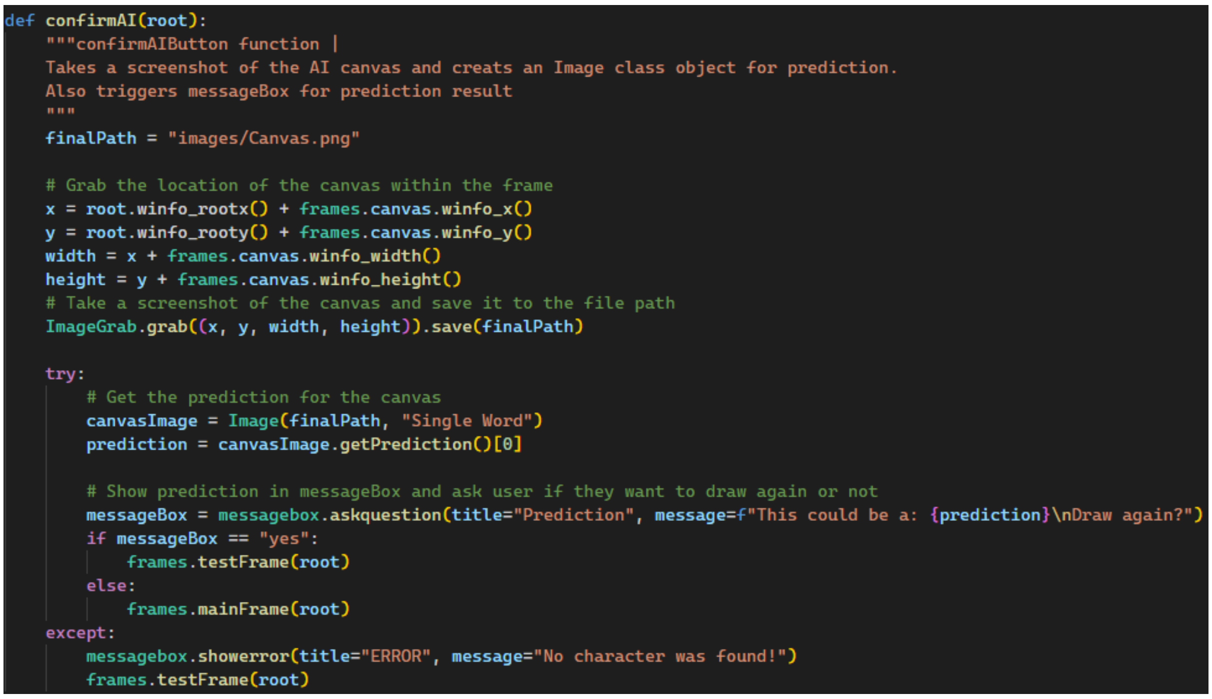
Fig. 42: TestAI class in its entirety

The TestAI class inherits the functionality of the tk.Canvas class for convenience's sake. To be able to draw lines in the position of the user’s cursor, the lastPos method is used. Otherwise, the addLine and delLine method respectively creates black and white lines for the user to draw with their left and right mouse buttons.



This could be a: A
Draw again?

Fig. 43: confirmAI pop up message box



```
def confirmAI(root):
    """confirmAIButton function |
    Takes a screenshot of the AI canvas and creats an Image class object for prediction.
    Also triggers messageBox for prediction result
    """
    finalPath = "images/Canvas.png"

    # Grab the location of the canvas within the frame
    x = root.winfo_rootx() + frames.canvas.winfo_x()
    y = root.winfo_rooty() + frames.canvas.winfo_y()
    width = x + frames.canvas.winfo_width()
    height = y + frames.canvas.winfo_height()
    # Take a screenshot of the canvas and save it to the file path
    ImageGrab.grab((x, y, width, height)).save(finalPath)

    try:
        # Get the prediction for the canvas
        canvasImage = Image(finalPath, "Single Word")
        prediction = canvasImage.getPrediction()[0]

        # Show prediction in messageBox and ask user if they want to draw again or not
        messageBox = messagebox.askquestion(title="Prediction", message=f"This could be a: {prediction}\nDraw again?")
        if messageBox == "yes":
            frames.testFrame(root)
        else:
            frames.mainFrame(root)
    except:
        messagebox.showerror(title="ERROR", message="No character was found!")
        frames.testFrame(root)
```

Fig. 44: confirmAI function

The confirmAIButton calls for this function when pressed. The confirmAI function takes a screenshot of the drawing canvas and saves it as a file in the images folder through the ImageGrab class' grab method from the Pillows module. The image will then be instantiated as an Image class object and the prediction from the image will be displayed through a TkInter pop up message box (more explanation about the Image class right after this). The TkInter pop up message box returns the user back to the testFrame if they click on yes, otherwise it will return the user back to mainFrame. Exception handling is used if no character was predicted by the AI model.

Image Processing Implementation

All the processing done for an image to be predicted is done inside the Image class in the image file. Here is its UML diagram:

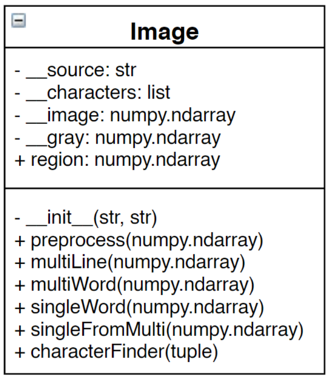
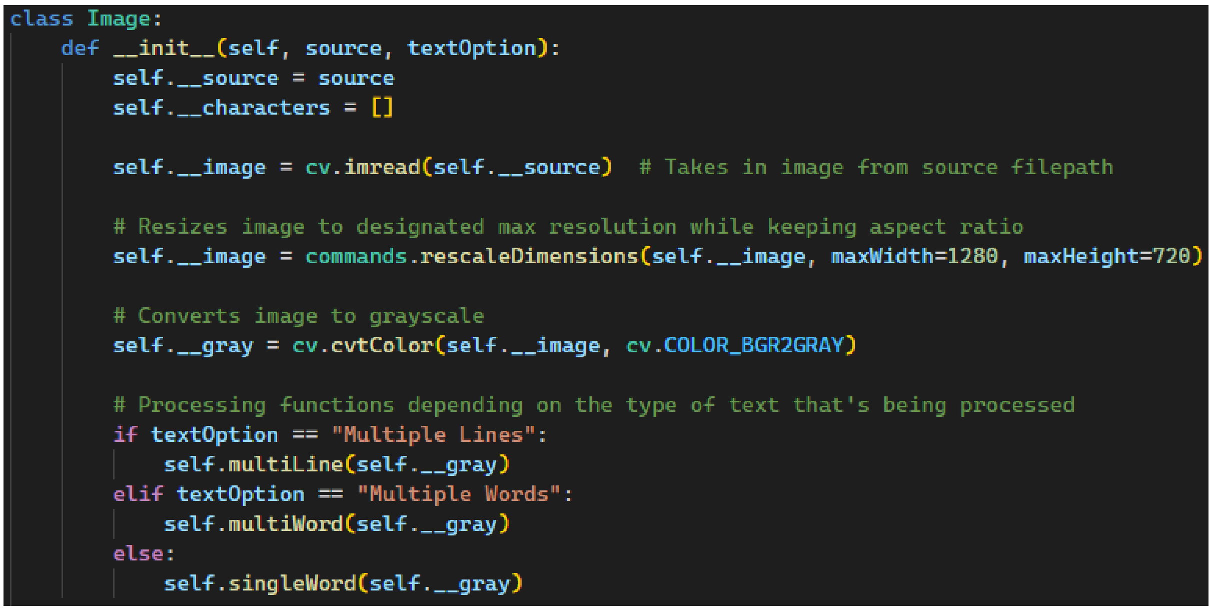


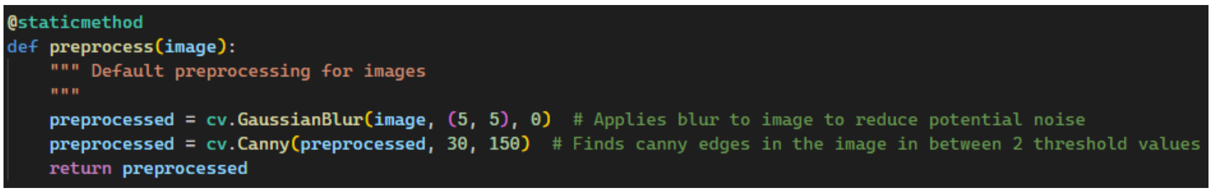
Fig. 45: Image class UML diagram



```python
class Image:
    def __init__(self, source, textOption):
        self.__source = source
        self.__characters = []

        self.__image = cv.imread(self.__source)  # Takes in image from source filepath

        # Resizes image to designated max resolution while keeping aspect ratio
        self.__image = commands.rescaleDimensions(self.__image, maxWidth=1280, maxHeight=720)

        # Converts image to grayscale
        self.__gray = cv.cvtColor(self.__image, cv.COLOR_BGR2GRAY)

        # Processing functions depending on the type of text that's being processed
        if textOption == "Multiple Lines":
            self.multiLine(self.__gray)
        elif textOption == "Multiple Words":
            self.multiWord(self.__gray)
        else:
            self.singleWord(self.__gray)
```

Fig. 46: Image class init

The Image class takes in source and textOption as its 2 parameters. source is the file path to where an image is stored while textOption is the type of text that the user chooses for the image. characters is a list that stores the prediction that the AI model makes after going through the processed image. All images that get passed in will have their dimensions resized to around 1280x720 so that they aren’t too small or too large. Depending on the textOption chosen previously, the grayscaled image will go through a different method that processes it.

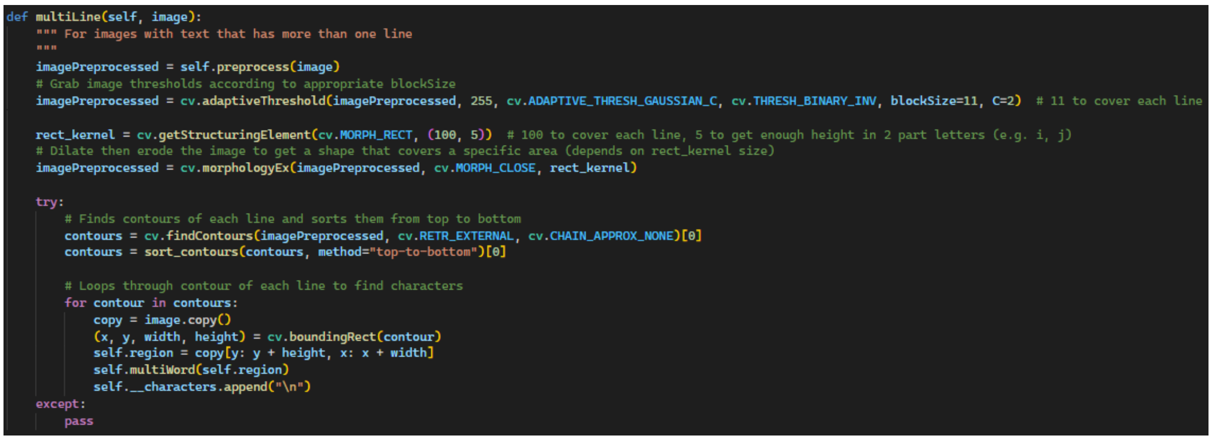


```python
@staticmethod
def preprocess(image):
    """ Default preprocessing for images
    """
    preprocessed = cv.GaussianBlur(image, (5, 5), 0)  # Applies blur to image to reduce potential noise
    preprocessed = cv.Canny(preprocessed, 30, 150)  # Finds canny edges in the image in between 2 threshold values
    return preprocessed
```

Fig. 47: Image class preprocess static method

Before processing the image, a preprocess method is called on the image almost every time when it goes through one of the methods. This method will perform a gaussian blur on the image to reduce unwanted noise and also finds the edges of the image both through OpenCV functions.

The complicated part comes when the image is being processed. Depending on the type of text that the image holds, a different method needs to be called because the image needs to be handled differently. As a result, 4 methods are utilized for this step.



```python
def multiLine(self, image):
    """ For images with text that has more than one line
    """
    imagePreprocessed = self.preprocess(image)
    # Grab image thresholds according to appropriate blockSize
    imagePreprocessed = cv.adaptiveThreshold(imagePreprocessed, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, blockSize=11, C=2)  # 11 to cover each line

    rect_kernel = cv.getStructuringElement(cv.MORPH_RECT, (100, 5))  # 100 to cover each line, 5 to get enough height in 2 part letters (e.g. i, j)
    # Dilate then erode the image to get a shape that covers a specific area (depends on rect_kernel size)
    imagePreprocessed = cv.morphologyEx(imagePreprocessed, cv.MORPH_CLOSE, rect_kernel)

    try:
        # Finds contours of each line and sorts them from top to bottom
        contours = cv.findContours(imagePreprocessed, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)[0]
        contours = sort_contours(contours, method="top-to-bottom")[0]

        # Loops through contour of each line to find characters
        for contour in contours:
            copy = image.copy()
            (x, y, width, height) = cv.boundingRect(contour)
            self.region = copy[y: y + height, x: x + width]
            self.multiWord(self.region)
            self.__characters.append("\n")
    except:
        pass
```

Fig. 48: Image class multiLine method

The multiLine method is for cases where the image contains more than one word in multiple lines, going from top to bottom. In this case, to classify the different lines, adaptive thresholding and closing are used. Adaptive thresholding determines the threshold for an image's pixels based on a small region around it instead of just globally. This in turn helps with cases where one part of an image is darker than the other. OpenCV's method of adaptive thresholding is used with a block size of 11 pixels to cover a larger area around the characters. Closing is the act of dilating and eroding an image. Adaptive thresholding creates small holes inside the characters which we will close up through closing. We can invoke closing through OpenCV's method of morphological transformations and specifying the specific image operation. Kernel size for that morphological transformation is the main differing value that differentiates the different methods inside the Image class. For larger objects like lines of text, a rectangular object of dimensions 100x5 is created to cover the words and spaces in between multiple words in a single line. As a result, each individual line is covered and accounted for, but not its individual characters yet.

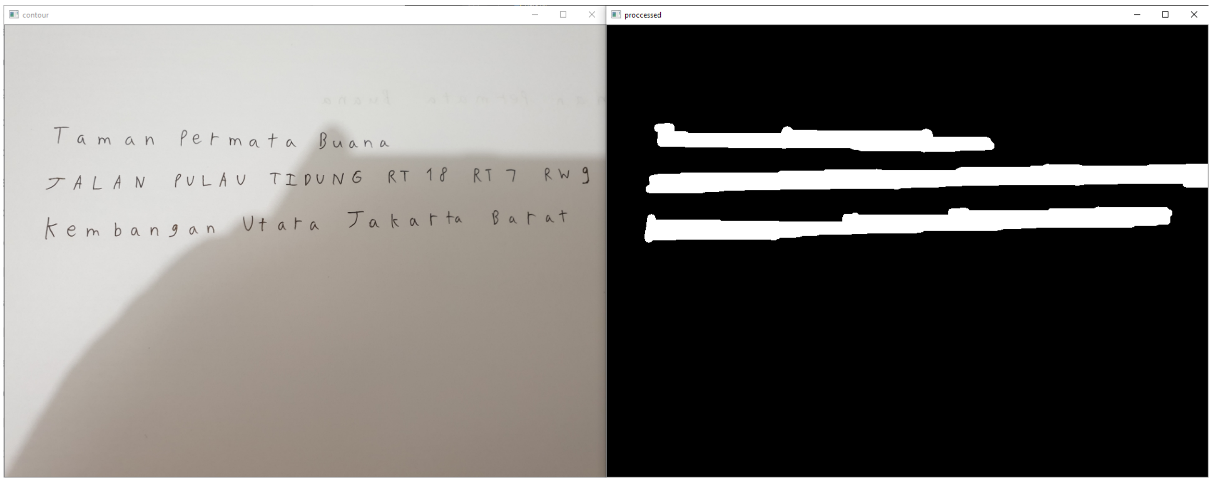
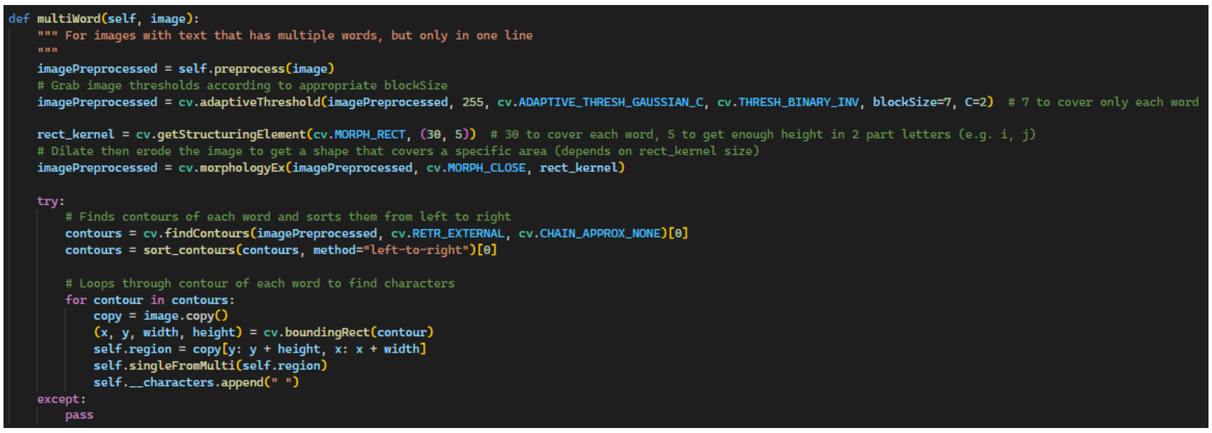

contour
proccessed
Taman Petmata Buana
JALAN PULAU TIDUNG RT 18 RT 7 RW 9
Kembangan Utara Jakarta Barat

Fig. 49: An example of how multiLine method works

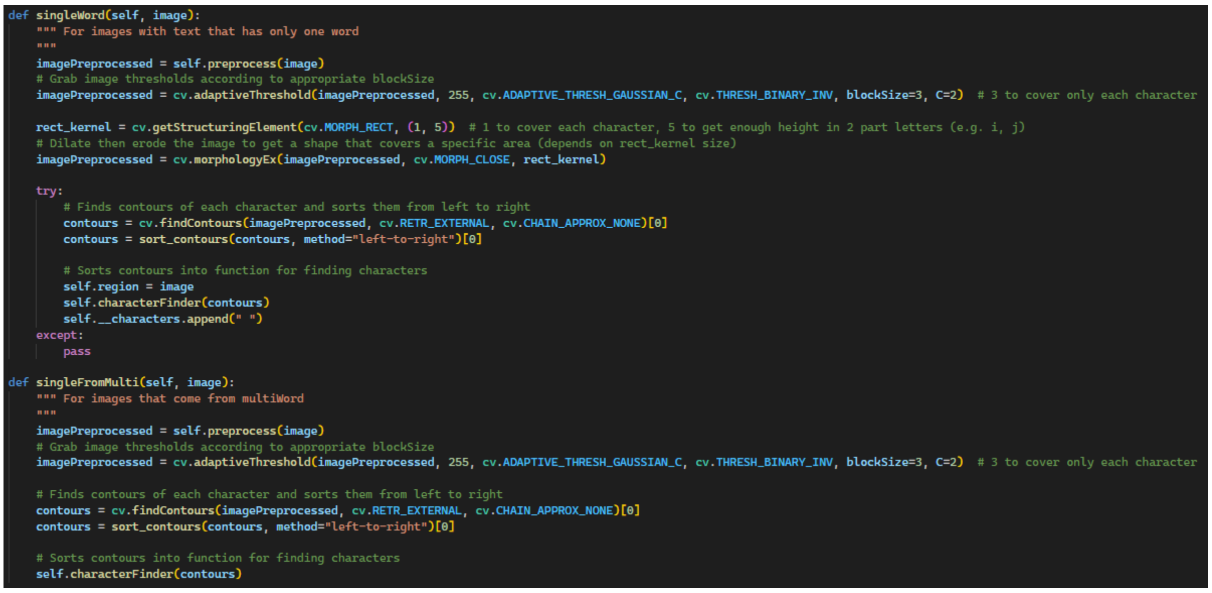
After the line processing, OpenCV's findContours function is used to differentiate the lines one by one and with the combination of the imutils module's sort_contours method, sorts the lines from top to bottom. Each line will then produce a copy of its covered region from the original passed through image, which will then be passed on to the multiWord method. After it goes through the other methods, a new line is appended to the characters list to indicate a line break.



```python
def multiWord(self, image):
    """ For images with text that has multiple words, but only in one line
    """
    imagePreprocessed = self.preprocess(image)
    # Grab image thresholds according to appropriate blockSize
    imagePreprocessed = cv.adaptiveThreshold(imagePreprocessed, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, blockSize=7, C=2)  # 7 to cover only each word

    rect_kernel = cv.getStructuringElement(cv.MORPH_RECT, (30, 5))  # 30 to cover each word, 5 to get enough height in 2 part letters (e.g. i, j)
    # Dilate then erode the image to get a shape that covers a specific area (depends on rect_kernel size)
    imagePreprocessed = cv.morphologyEx(imagePreprocessed, cv.MORPH_CLOSE, rect_kernel)

    try:
        # Finds contours of each word and sorts them from left to right
        contours = cv.findContours(imagePreprocessed, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)[0]
        contours = sort_contours(contours, method="left-to-right")[0]

        # Loops through contour of each word to find characters
        for contour in contours:
            copy = image.copy()
            (x, y, width, height) = cv.boundingRect(contour)
            self.region = copy[y: y + height, x: x + width]
            self.singleFromMulti(self.region)
            self.__characters.append(" ")
    except:
        pass
```

Fig. 50: Image class multiWord method

The multiWord method is generally the same as the multiLine method, except that it uses a smaller adaptive threshold block size of 7 and a smaller kernel size of 30x5 for the morphological transformation. This is to prevent spaces between words from being counted when it's passed through the findContours method. The other difference is that the imutils' sort_contours method is used with the left-to-right method instead of top to bottom. Afterwards, the same loop happens for the words found, which is passed on to the singleFromMulti method. After it goes through the other methods, a whitespace is appended to the characters list.

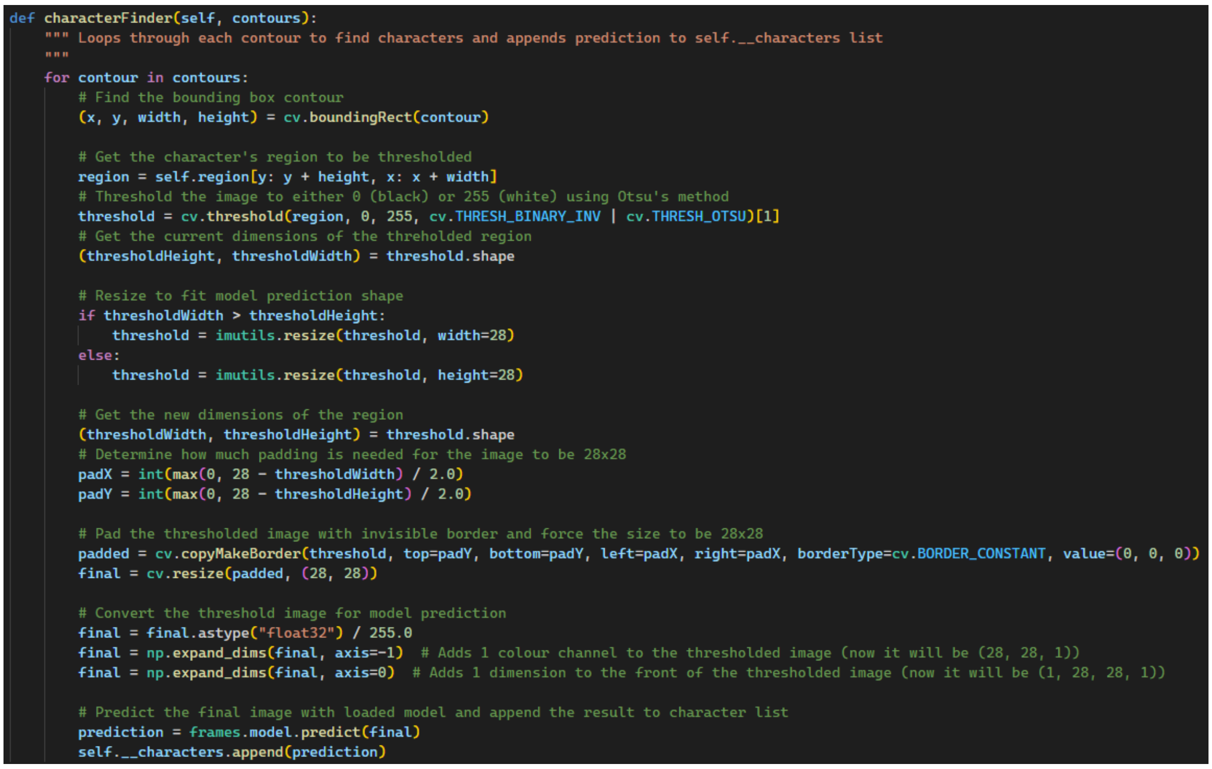


```python
def singleWord(self, image):
    """ For images with text that has only one word
    """
    imagePreprocessed = self.preprocess(image)
    # Grab image thresholds according to appropriate blockSize
    imagePreprocessed = cv.adaptiveThreshold(imagePreprocessed, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, blockSize=3, C=2)  # 3 to cover only each character

    rect_kernel = cv.getStructuringElement(cv.MORPH_RECT, (1, 5))  # 1 to cover each character, 5 to get enough height in 2 part letters (e.g. i, j)
    # Dilate then erode the image to get a shape that covers a specific area (depends on rect_kernel size)
    imagePreprocessed = cv.morphologyEx(imagePreprocessed, cv.MORPH_CLOSE, rect_kernel)

    try:
        # Finds contours of each character and sorts them from left to right
        contours = cv.findContours(imagePreprocessed, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)[0]
        contours = sort_contours(contours, method="left-to-right")[0]

        # Sorts contours into function for finding characters
        self.region = image
        self.characterFinder(contours)
        self.__characters.append(" ")
    except:
        pass

def singleFromMulti(self, image):
    """ For images that come from multiWord
    """
    imagePreprocessed = self.preprocess(image)
    # Grab image thresholds according to appropriate blockSize
    imagePreprocessed = cv.adaptiveThreshold(imagePreprocessed, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY_INV, blockSize=3, C=2)  # 3 to cover only each character

    # Finds contours of each character and sorts them from left to right
    contours = cv.findContours(imagePreprocessed, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)[0]
    contours = sort_contours(contours, method="left-to-right")[0]

    # Sorts contours into function for finding characters
    self.characterFinder(contours)
```

Fig. 51: Image class singleWord and singleFromMulti method

The singleWord and singleFromMulti differ in the fact that the singleWord still does the morphological transformation for the image, albeit at a much smaller size of 1x5. This is because images that come from multiWord are very small in size and can cause errors when processing again with morphological transformation. As a result, 2 different methods were created for single words.

After both are done processing, the characters that are found go through the final processing method, characterFinder.

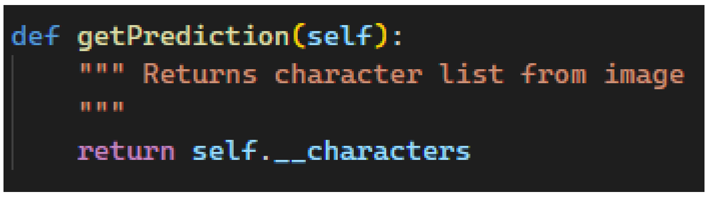


```python
def characterFinder(self, contours):
    """ Loops through each contour to find characters and appends prediction to self.__characters list
    """
    for contour in contours:
        # Find the bounding box contour
        (x, y, width, height) = cv.boundingRect(contour)

        # Get the character's region to be thresholded
        region = self.region[y: y + height, x: x + width]
        # Threshold the image to either 0 (black) or 255 (white) using Otsu's method
        threshold = cv.threshold(region, 0, 255, cv.THRESH_BINARY_INV | cv.THRESH_OTSU)[1]
        # Get the current dimensions of the threholded region
        (thresholdHeight, thresholdWidth) = threshold.shape

        # Resize to fit model prediction shape
        if thresholdWidth > thresholdHeight:
            threshold = imutils.resize(threshold, width=28)
        else:
            threshold = imutils.resize(threshold, height=28)

        # Get the new dimensions of the region
        (thresholdWidth, thresholdHeight) = threshold.shape
        # Determine how much padding is needed for the image to be 28x28
        padX = int(max(0, 28 - thresholdWidth) / 2.0)
        padY = int(max(0, 28 - thresholdHeight) / 2.0)

        # Pad the thresholded image with invisible border and force the size to be 28x28
        padded = cv.copyMakeBorder(threshold, top=padY, bottom=padY, left=padX, right=padX, borderType=cv.BORDER_CONSTANT, value=(0, 0, 0))
        final = cv.resize(padded, (28, 28))

        # Convert the threshold image for model prediction
        final = final.astype("float32") / 255.0
        final = np.expand_dims(final, axis=-1)  # Adds 1 colour channel to the thresholded image (now it will be (28, 28, 1))
        final = np.expand_dims(final, axis=0)  # Adds 1 dimension to the front of the thresholded image (now it will be (1, 28, 28, 1))

        # Predict the final image with loaded model and append the result to character list
        prediction = frames.model.predict(final)
        self.__characters.append(prediction)
```

Fig. 52: Image class characterFinder method

The characterFinder method goes through each character that it finds from the image and creates another copy of just that specific character from the previously cropped region. A global threshold is used to the image, which binarizes it to just either pure black or pure white through Otsu's thresholding method.

The current dimensions of the image is then taken to determine either to resize it from its width or height. With the new dimensions of the image, padding is applied to ensure that its size will definitely be 28x28. Finally, before passing it on to the AI Model, the image is converted into float32 and divided by 255. Its dimensions are also expanded to include 1 colour channel and another dimension for batch size (of 1). The image will then be fed into the model that was loaded in the beginning of the program and the prediction for that character is appended to the characters list.

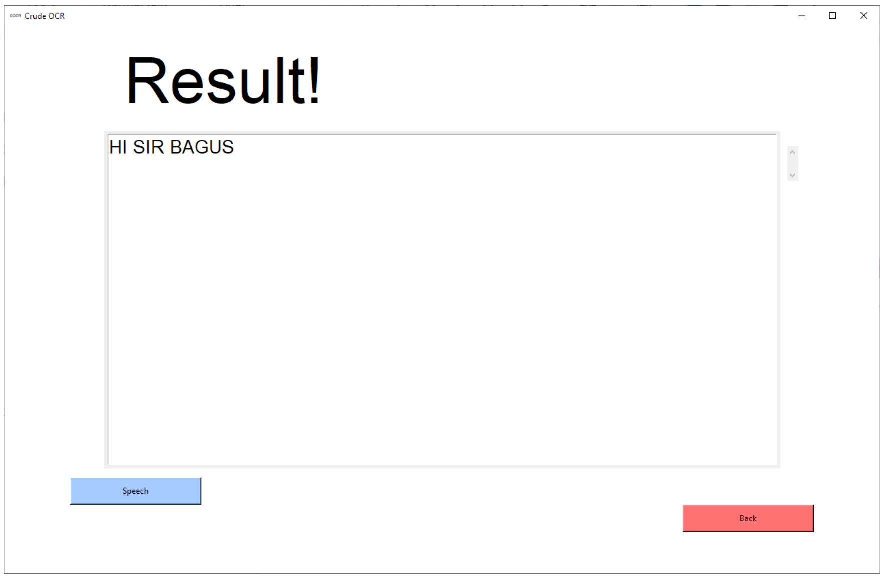


```python
def getPrediction(self):
    """ Returns character list from image
    """
    return self.__characters
```

Fig. 53: Image class getPrediction method

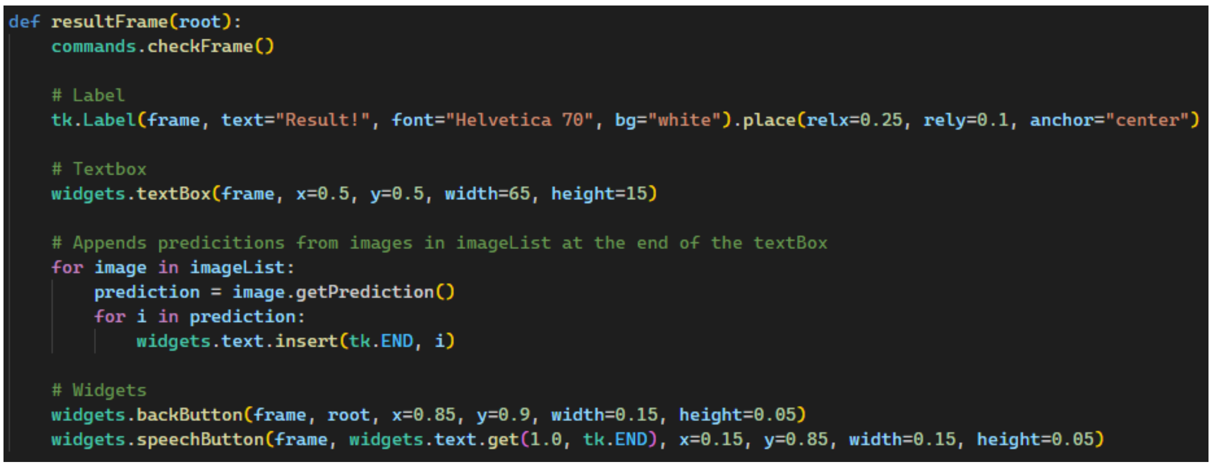
Finally, other parts of the program can access the characters list through the getPrediction method.

Result Frame



HI SIR BAGUS

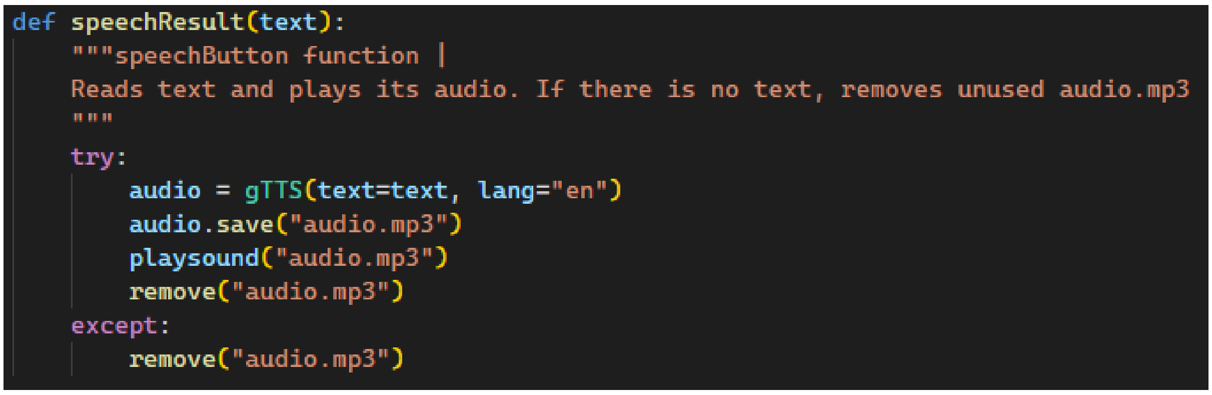
Fig. 54: Crude OCR resultFrame



```python
def resultFrame(root):
    commands.checkFrame()

    # Label
    tk.Label(frame, text="Result!", font="Helvetica 70", bg="white").place(relx=0.25, rely=0.1, anchor="center")

    # Textbox
    widgets.textBox(frame, x=0.5, y=0.5, width=65, height=15)

    # Appends predicitions from images in imageList at the end of the textBox
    for image in imageList:
        prediction = image.getPrediction()
        for i in prediction:
            widgets.text.insert(tk.END, i)

    # Widgets
    widgets.backButton(frame, root, x=0.85, y=0.9, width=0.15, height=0.05)
    widgets.speechButton(frame, widgets.text.get(1.0, tk.END), x=0.15, y=0.85, width=0.15, height=0.05)
```

Fig. 55: resultFrame function

The final part of the program is the resultFrame. With the textBox function that's called, all the images in the imageList gets its getPrediction method called. The characters in the prediction object are then inserted to the end of the textBox.



```python
def speechResult(text):
    """speechButton function |
    Reads text and plays its audio. If there is no text, removes unused audio.mp3
    """
    try:
        audio = gTTS(text=text, lang="en")
        audio.save("audio.mp3")
        playsound("audio.mp3")
        remove("audio.mp3")
    except:
        remove("audio.mp3")
```

Fig. 56: speechResult function

The speechButton widget calls for this command when pressed. This invokes the gTTS module’s gTTS function, which takes in the text inside of textBox and produces an mp3 file from it. The playsound module’s playsound function is then called to play the audio file produced. After that, the audio file is removed using the os module’s remove function.

References

- Flowchart designer
  - app.diagrams.net
- TensorFlow and Machine Learning
  - pyimagesearch.com/2020/08/17/ocr-with-keras-tensorflow-and-deep-learning/
  - towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
  - tensorflow.org/datasets/catalog/emnist
  - youtube.com/watch?v=VwVg9jCtqaU
- TkInter
  - docs.python.org/3.8/library/tk.html
  - pyimagesearch.com/2016/05/30/displaying-a-video-feed-with-opencv-and-tkinter/
  - tutorialspoint.com/python/python_gui_programming.htm
- Image Processing
  - github.com/jrosebr1/imutils
  - docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html
  - docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html
  - docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html
  - tutorialkart.com/opencv/python/opencv-python-resize-image/