Hieroglyphics

Data Science 440
Project Report (to customer)

Team Six

Juliana(Jiayin) Hu, Sydney Wehn, Kangdong Yuan, Xueqing Zhang, Yuqi Gao

Table of Contents:

Background

At the beginning of the semester, our team was challenged with the task of making sense of hieroglyphics for our client Dr. Redford. During our early meetings, we found that the customer wanted us to use our data science backgrounds to help with scribe identification. Our group struggled at the beginning because we were not provided a dataset. Therefore, we spent time exploring potential options and found the GitHub dataset below.
https://github.com/yedkk/ds440-r/tree/main/DS440/Dataset/Manual/Preprocessed

After determining that we would go ahead with this dataset, we branched off as a group to work on clustering, classification, and cartouche examination. These methods all have individual contributions to our overall goal of making sense of hieroglyphics. K-means and feature extraction clustering provide insights as to how images can be grouped together to represent 26 Gardiner Sign categories within the hieroglyphic alphabet. ResNet34 and AlexNet models were also created using the same dataset of 4,210 images to see how different layers classified these images, based on features of pixels, into the same 27 classes mentioned before. Cartouche research and investigation was also done to align our project with the client's vision. With all of these different data science methods, we aim to deliver knowledge to the customer about how they can feed in image hieroglyphic data to then be clustered and classified into Gardiner Sign categories with image recognition. We also aim to provide insights into how our models performed and how they can be translated to additional datasets. Additionally, with the cartouche recognition, we will add another aspect of scribe identification to further our client's knowledge of how different scribes played major roles in the world of hieroglyphics.

Methods

Feature Extraction (contributed by Kangdong Yuan)

In order to perform k-means clustering training. The first thing I need to do is to build a deep learning environment. I decided to use python's anaconda platform for my deep learning program. Setting up the environment took me a lot of time, because the TensorFlow environment requires a large number of third-party libraries to support, and these environments have special requirements for the python and anaconda versions . After setting up the execution environment, I imported a lot of python third-party libraries that need to be used.

Import the packages for future use

```
[ ]  from keras.preprocessing import image
     from keras.preprocessing.image import load_img
     from keras.preprocessing.image import img_to_array
     from keras.applications.vgg16 import preprocess_input
     from PIL import Image as pil_image
     from keras.applications.vgg16 import VGG16
     from keras.models import Model
     from sklearn.cluster import KMeans
     from sklearn.decomposition import PCA
     import os, shutil, glob, os.path
     import numpy as np
     import matplotlib.pyplot as plt
     from random import randint
     import pandas as pd
     import pickle
     import torch
```

After obtaining thousands of hieroglyphic pictures without any markings, I first cleaned up these pictures and used python's NumPy and matplotlib (third-party libraries) to remove the pictures whose resolution and color did not meet the standard. But this step is optional when your dataset meets the standard.

```
def match_image_by_color(image, color, threshold=60, number_of_colors=10):
    image_colors = get_colors(image, number_of_colors, False)
    selected_color = rgb2lab(np.uint8(np.asarray([[color]])))

    select_image = False
    for i in range(number_of_colors):
        curr_color = rgb2lab(np.uint8(np.asarray([[image_colors[i]]])))
        diff = deltaE_cie76(selected_color, curr_color)
        if (diff < threshold):
            select_image = True

    return select_image
```

Then I put more than 4000 images to be processed in a folder, and I created a result storage folder to store the clustered images. Through a lot of information search, I learned that if I want to extract the features of each image, I need to convert the different pixels of each image into

grayscale numbers. Then I need to convert each picture into a matrix of gray values. Then I extracted the features of each image from these matrices, and added these features to a list purely for deep learning training.

Extract the feature to the list

```
[ ]  filelist = glob.glob(os.path.join(imdir, '*.png'))
     # filelist.sort()
     featurelist = []
     for i, imagepath in enumerate(filelist):
         print("    Status: %s / %s" %(i, len(filelist)), end="\r")
         img = image.load_img(imagepath, target_size=(224, 224))
         img_data = image.img_to_array(img)
         img_data = np.expand_dims(img_data, axis=0)
         img_data = preprocess_input(img_data)
         features = np.array(model.predict(img_data))
         featurelist.append(features.flatten())
```

K-Means Clustering (contributed by Kangdong Yuan)

After all the image features were extracted, I started to use kmean to train my deep learning model with a number of clusters equal to 27. Then, through the fitting process of this model, the pictures classified by the k-means model are saved in my results folder. The label of each cluster is identified by filename.

```
number_clusters = 27
kmeans = KMeans(n_clusters=number_clusters,
random_state=0).fit(np.array(featurelist))
```

When I was training my k-means model, I found that my training speed was particularly slow, and my cpu usage had been declining since the start of execution. In practical applications, I need to find a more efficient way to train my model. When I was training my k-means model, I found that my training speed was particularly slow, and my cpu usage had been declining. In practical applications, I need to find a more efficient way to train my model. Through my exploration of the matrix, I found that the matrix of each image has 224 numbers. If I use such a large matrix to train my k-means model, the cpu will face tremendous pressure and even cause the temperature to be too high and unstable. So I used the PCA method to reduce the size of the matrix. PCA is a mathematical method to convert data points in high dimensionality into data points in low dimensionality. I use the PCA library of python to call the PCA function. When I reduced the matrix dimension of the training data to 10, the training speed of my k-means model dropped from the half hour to 20 seconds. Through the PCA method, I improved the efficiency of model training to a satisfactory level.

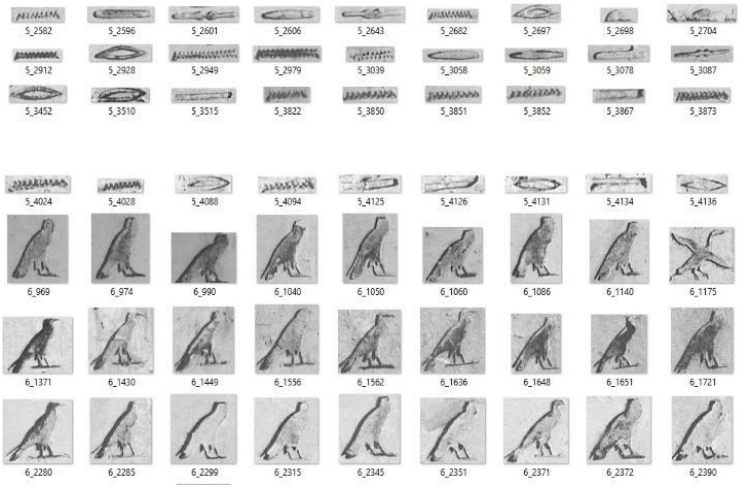
Do the PCA to reduce the dismension of matrix, I suggest user set n_digits between 10-20

```
[ ]  n_digits=10

     pca = PCA(n_components=n_digits).fit_transform(featurelist)
```

Do the Kmeans training, user can change the number_clusters freely

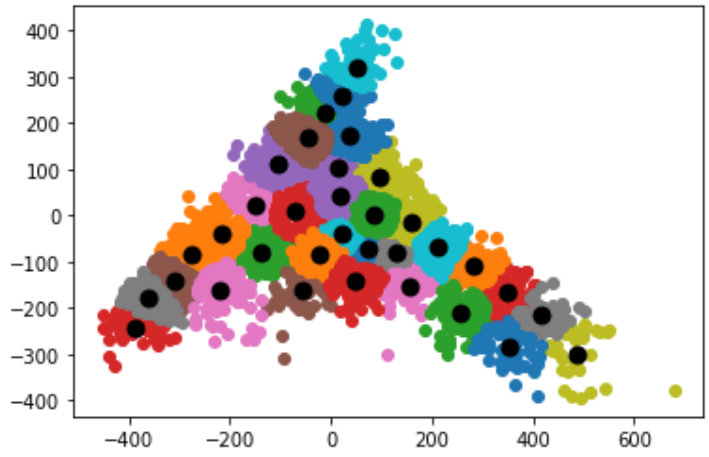
```
[ ]  number_clusters=27

     kmeans = KMeans(n_clusters=number_clusters, random_state=0)
     kmeans.fit(pca)

     KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
            n_clusters=27, n_init=10, n_jobs=None, precompute_distances='auto',
            random_state=0, tol=0.0001, verbose=0)
```

I double-checked my results folder, and I think my k-means classification has achieved a certain degree of success, because each group of images has a large number of similarities. Although the similarity of some images in the same group is not very high, I suspect that these errors are due to my mistakes in selecting the number of clusters. So, I need to test multiple times and consult the hieroglyphic file to determine the exact number of clusters. Moreover, I also want to change my parameters of kmean to get better results.

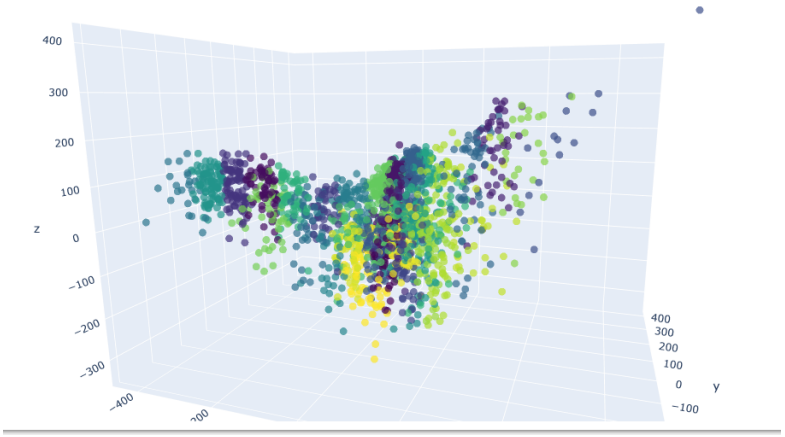


Visualization (contributed by Kangdong Yuan)

In the end, when I need to verify the accuracy of my k-means model, in addition to verifying by viewing the results, we can also verify by visualizing the k-means model. First, I extract the labels of each data point after applying the k-means model. Then in order to show the data points in plane and three-dimensional coordinates, I used the PCA method to convert ten-dimensional data points into two-dimensional and three-dimensional data points. Then I use the two-dimensional and three-dimensional data as the spatial coordinates of each data point to draw the graph. Finally, for each data point in a different cluster, I use a unique color to represent it.



Because my data has four thousand data points, I want to use interactive models to make my three-dimensional images. So I used the plotly library to build an interactive three-dimensional coordinate system, and then represented each three-dimensional data point in this coordinate system and used color to represent the classification of each data point.



ResNet34 and AlexNet Classification (contributed by Sydney)

After determining where we were headed with our hieroglyphics project, we branched off as a group and decided that we would focus on both supervised and unsupervised learning through different clustering and classification techniques. Personally, I chose to focus on supervised classification. To do so, I decided to implement ResNet and AlexNet models on the 27 classes outlined in the Gardiner Sign list. As for the environment of the code, I chose Google Colabs. I set up directories based on the names of the photos which correspond to the Egyptian hieroglyphic alphabet. After a few runtimes, I saw that my GPU on regular Google Colabs was

not sufficient for processing data this large. This is why I upgraded to Google Colab Pro which allowed me to run the models with higher GPU.

Once my environment was set up, I decided to try ResNet18, 34, and 152. ResNet34 performed the best so I proceeded with this model and created data loaders to transform the data using random cropping and flipping methods for processing. I also implemented the pretrained model AlexNet and used the same transformations as ResNet34. I then went ahead and tuned parameters associated with both models after endless hours of researching and looking into documentation on these models.

ResNet34 is a residual learning framework that efficiently trains networks that are deep and more complex than previous layers. They use repetitive reformulating of layers with previous reference layer inputs instead of learning with new functions everytime.

AlexNet consists of five convolutional layers and three fully connected layers. The activation used is Rectified Linear Unit (RELU) to help with vanishing gradient problems. With vanishing gradient problems, each of the neural network's weights receive and make updates proportionate to the partial derivative of the error function and its weights at each iteration. With this, we must ensure that its gradients do not vanish too small that they do not make proportional differences. With AlexNet, data augmentation also results in a reduction of overfitting and introduces drop outliers to drop connections with probabilities of 0.5 or smaller.

After learning about the specific architectures and implications of these two models, I also researched drop out, normalization, and scheduling techniques, for both models, which allowed me to successfully process and run the datasets to classify which labels the model believed each image belonged to. This is a type of supervised learning as I had the correct labels for each photo and trained my models without the knowledge of these previously known labels.

I then ensured that both models were outputting 27 classification groups and used SGD optimizers and StepLR schedulers to decay the learning rate by a factor of 0.001 every 11 epochs. Then I trained the models and created lists of training and validation losses to then create confusion matrices that allowed me to understand how my models were performing in terms of precision, accuracy, and recall.

When tuning and evaluating my model, I first had very low accuracies in the range 35-45%. Then when I changed the learning rate, worked with drop out rates, and shuffling techniques, I was running the model with higher accuracy. The highest accuracy for ResNet was 69.4290% while AlexNet was 59.1028%.

The optimal parameters for ResNet were learning rate = 0.001, momentum = 0.90, step size = 7, and epochs = 11. The best parameters for AlexNet were learning rate = 0.001, momentum = 0.90, step size = 7, and epochs = 8. I found that when running other datasets in my models that had color, I had higher accuracies due to the fact that RBG image processing and classification results in less misclassification as colors make it possible to pull key features of images.

Overall, this project has challenged me in many ways. As team leader, I was challenged with the fact that one of our team members is in China, which had a strong influence on our meeting times and ways of collaboration. I also ran into bottlenecks associated with trying to report to the professor our progress when team members were not responding, attending meetings, and updating their work. This put time constraints on my individual work as I was always spending time submitting, reporting, and editing the work of others in addition to that of my own. On a technical aspect, I learned a lot about how local and virtual machines vary in computing power and package versions. I read a lot of code documentation and debugging forums which also gave me more experience with specific problem solving in Python. Before this project, I also did not know anything about transfer learning or AlexNet and had very limited knowledge of what residual networks could do. After working with these two types of models, I gained more experience and insight as to how learning rates, optimizers, and dropouts had strong impacts on neural networks and their classification performance.

Additionally, due to the fact that we were not provided a dataset, the way that I set up the directories was not optimal. I had to manually drag all of the images into their respective folders in the training, testing, and validation sets. Not only was this a burden time wise, but it also was not as accurate of a split as a typical train test split with designated ratios would be. So when moving forward, I would recommend to our client that they try to use a dataset that has directories and associated images already set up.

Cartouche (Contributed by Yuqi)

I started with doing some sort of the experiment on shape recognizing, but the result did not turns out well, I had tried the contour measure, however, it works better on simple shapes, however, cartouche was more complex, since they did had a basic shape, but the length and the area etc. these values are basically different from each others, so the contour measure method went into the drain.

And for later I tried the template matching method, which however, also does not work well. It does work well on static regular shapes or components recognition, thus, theoretically, if I picked a component from the motherboard of a PC, and it will be easy to use this method to recognize all the other same or duplicated components on the motherboard. But for irregular shapes, it works poorly.

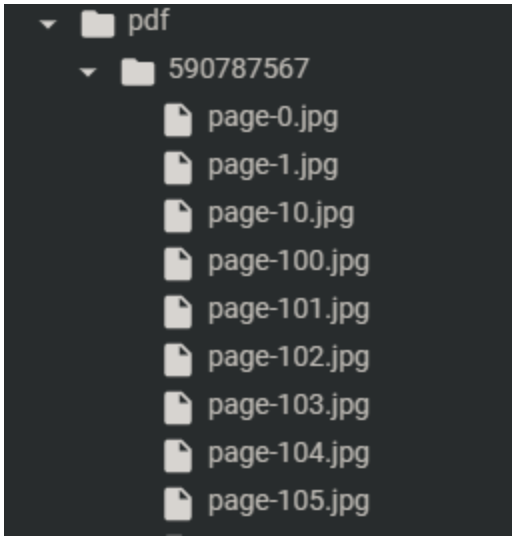
Until here, my attempts to try to bypass the complex model training method was gone. Then I realize that to build a dataset myself is inevitable, and use that self made dataset to train a model to do the cartouche recognition.

So the first step is to source out some cartouche examples online. And turns out there are plenty of records which have been digitized and published on the Internet if you dig hard enough through some archive websites. After using hours to look through each archive with the topic including the keyword “hieroglyph”, because some of them included a lot of graphs and records from the original slab, some of them are just heavy text-based research written in books. I picked the ones which are heavy graph-based. And because they are online PDFs, which I head on the research on how to scrape the pictures from the PDFs for future use. While using the package PyMuPDF allows me to achieve this work pretty successfully, the downside is I still need to download the PDF files to local directories for future use.

```
    print('extract all pdf to images(jpg): {}'.format(pdf_dir))

    if not os.path.exists(pdf_dir):
        print('pdf dir not exists: {}'.format(pdf_dir))
        return

    # Get a list of all PDF file names
    pdf_names = os.listdir(pdf_dir)
    pdf_names = list(filter(lambda name: name.endswith('.pdf'), pdf_names))

    if not pdf_names:
        print('pdf file not found')
        return

    cnt = 1
    total = len(pdf_names)
    percent = calc_percent(cnt, total)
    print('{}% -> {}/{}'.format(percent, cnt, total))

    # Extract one by one
    for pdf_name in pdf_names:
        # PDF file name
        pdf_filename = os.path.join(pdf_dir, pdf_name)

        # Image output folder name
        img_output_dir = pdf_name.replace('.pdf', '')
        img_output_dir = os.path.join(pdf_dir, img_output_dir)

        # Delete the old picture output folder
        if os.path.exists(img_output_dir):
            shutil.rmtree(img_output_dir)

        # Create a new image output folder
        os.makedirs(img_output_dir)

        # Open PDF files
        with fitz.open(pdf_filename) as doc:
            page_cnt = 0
            page_total = doc.page_count

            # Traverse all pages
            for page in doc:
                img_filename = os.path.join(img_output_dir, 'page-{}.jpg'.format(page.number))

                # Get the bitmap of the page
                pix = page.get_pixmap()

                # Save the bitmap as a file
                img = Image.frombytes('RGB', [pix.width, pix.height], pix.samples)
                img.save(img_filename, 'JPEG')

                page_cnt += 1
                page_percent = calc_percent(page_cnt, page_total)
                print('\rpdf: {}% -> {}/{}, page: {}% -> {}/{}'.format(percent, cnt, total, page_percent, page_cnt, page_total), end='')

        # Print the progress
        percent = calc_percent(cnt, total)
        cnt += 1
        print('{}% -> {}/{}'.format(percent, cnt, total))
```

```
pdf_dir = 'drive/MyDrive/pdf'
extract_images_from_pdf(pdf_dir)
```

Then I defined a function to do a series of processes to scrape the images and save them in a defined directory. And output result looks like this:



Then, What I need to do is to label this images, I used the tool called LabelImg, which in this tool, user can label a specific area of the image and define or name that area in a word, for example, in a photo of an apple is surrounded by bananas, the user can crop that apple's location out, and label that as "apple", so later the model knows what object in that area is. I was able to label some of the images, since there are a lot of images, labelling them by myself will take a lot of time than expected, then the LabelImg tool can output the file as an .xml file for future use, however, one problem I noticed in later when training the model, because I did the work both on local machine and the Google Colab, and when the .xml files are reading the location or the path of the image they belongs to, I need to manually change the path in the .xml files, there might be a better way, but despite the time, I left that along, since I already had a trained model under a local environment.

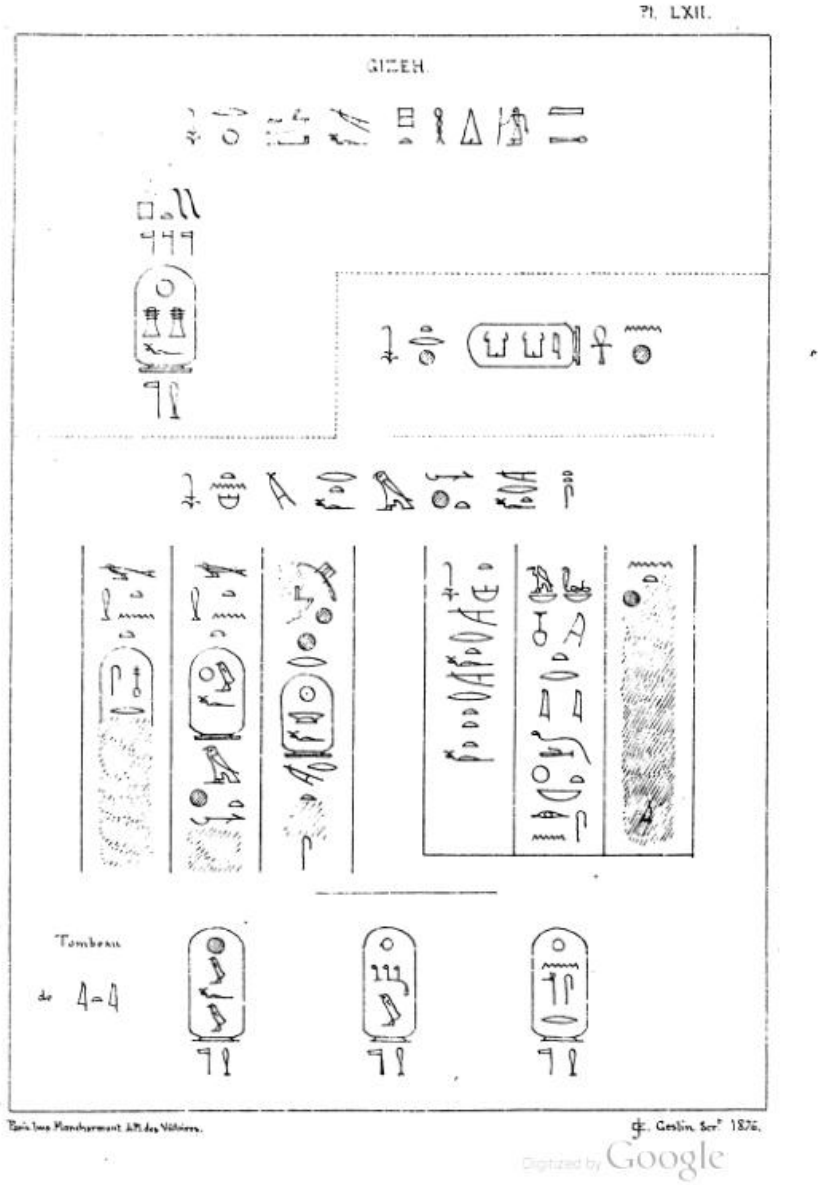
And at last, I decided to use imageai package to define the trainer for the model, and using tensorflow to use deep learning to do the training job

```python
import tensorflow as tf
from imageai.Detection.Custom import DetectionModelTrainer


def train():
    trainer = DetectionModelTrainer()
    trainer.setModelTypeAsYOLOv3()
    trainer.setDataDirectory(data_directory=os.path.join('drive','MyDrive','train_data', 'circle'))
    trainer.setTrainConfig(
        object_names_array=['circle'],
        batch_size=4,
        num_experiments=200,
        train_from_pretrained_model='pretrained-yolov3.h5'
    )
    trainer.trainModel()


def init_tf():
    # Set up GPU video memory to apply on demand to prevent insufficient video memory
    gpus = tf.config.experimental.list_physical_devices('GPU')
    if gpus:
        try:
            # Currently, memory growth needs to be the same across GPUs
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
            logical_gpus = tf.config.experimental.list_logical_devices('GPU')
            print(len(gpus), "Physical GPUs,", len(logical_gpus), "Logical GPUs")
        except RuntimeError as e:
            # Memory growth must be set before GPUs have been initialized
            print(e)

    print_is_gpu_available()


def print_is_gpu_available():
    print('gpu available: {}'.format(tf.test.is_gpu_available()))
```

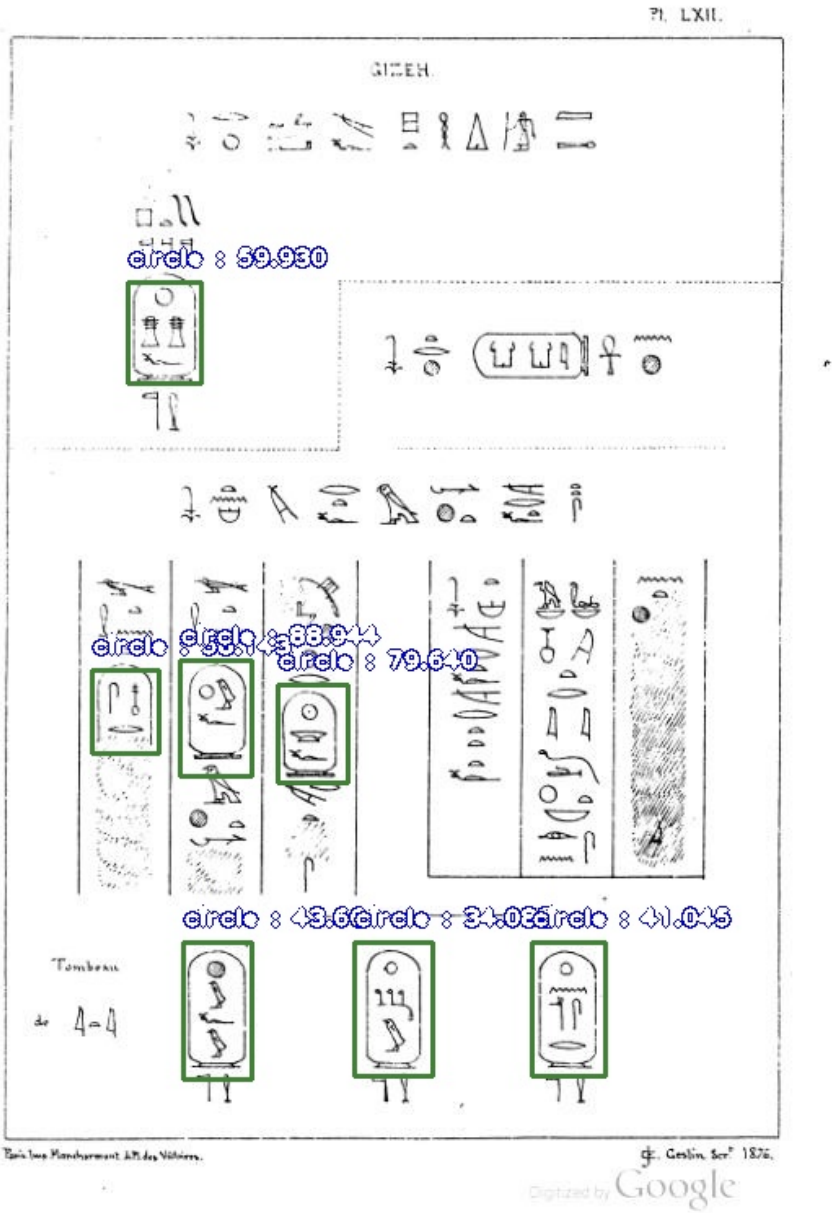
To be mentioned I used the pretrained-yolov3.h5:
https://github.com/OlafenwaMoses/ImageAI/releases/tag/essential-v4
Which is aYOLOv3 model for transfer learning when training new detection models in the imageAI.
And I took the model which has the lowest loss as the detection model which is around 14 at the moment

```python
from imageai.Detection.Custom import CustomObjectDetection

def test(img_path, img_out, minimum_percentage_probability=10):
    """
    Recognition test
    :param img_path \\Image file path
    :param img_out \\Result image path
    :param minimum_percentage_probability \\Minimum similarity of graphics
    """
    print('test: {} -> {}'.format(img_path, img_out))

    # Load the model
    detector = CustomObjectDetection()
    detector.setModelTypeAsYOLOv3()
    detector.setModelPath('drive/MyDrive/train_data/circle/models/detection_model-ex-138--loss-0014.575.h5')
    detector.setJsonPath('drive/MyDrive/train_data/circle/models/detection_config.json')
    detector.loadModel()

    # Detect
    detections = detector.detectObjectsFromImage(
        input_image=img_path,
        output_image_path=img_out,
        minimum_percentage_probability=minimum_percentage_probability
    )

    # Print the result
    for detection in detections:
        print("{}: {} -> {}".format(detection['name'], detection['percentage_probability'], detection['box_points']))
```

And I take this image as the test image:



And the output after the detection looks pretty promising:



The attempt of this experiment is trying to help with the study of the egyptian hieroglyphs, of course, with a better trained model than this, the potential of this concept will go way further, there are also another package called Detecto, however, that works better with pre-recorded videos or live-cam streaming videos, with this concept, if more people can contribute to label the cartouches or even the hieroglyphs in the real field in the future and trains a way better model, it might be possible for the archaeologists and researchers do an AR scan to detect the information in real time on the slabs or any discoveries.

Overall, the project was a success as we were able to deliver multiple working clustering and classification models. There was definitely a disconnect throughout this project as we did not have a real client. This posed challenges throughout the project as we were setting our own goals

rather than meeting and working with those of the client. However, with accuracies as high as 69% with ResNet and impurities of 4.5 with clustering techniques, we were able to provide value.

Our work is transferable to future work as other researchers could input their datasets into our clustering and classification models to create findings for their projects. Additionally, future researchers could work with our current code to try to improve performance. Another way that our work could be used in the future would be researchers using hieroglyphics data. In this case, they could look at what we have done to determine findings about the Gardiner Sign data and how it is distributed across clusters and different classes.

Works Cited (contributed by Sydney and Kangdong)

Tensorflow team. "Build from Source : Tensorflow," May 15, 2017. https://www.tensorflow.org/install/source.

Patel, Khush. "Architecture Comparison of AlexNet, VGGNet, ResNet, Inception, DenseNet." *Medium*, Towards Data Science, 8 Mar. 2020, towardsdatascience.com/architecture-comparison-of-alexnet-vggnet-resnet-inception-densenet-beb8b116866d.

Real Python. "Three Ways of Storing and Accessing Lots of Images in Python." *Real Python*, Real Python, 19 Mar. 2021, realpython.com/storing-images-in-python/.

"ResNet, AlexNet, VGGNet, Inception: Understanding Various Architectures of Convolutional Networks." *CV*, 9 Aug. 2017, cv-tricks.com/cnn/understand-resnet-alexnet-vgg-inception/.

"Scribes in Ancient Egypt." *Ancient Egypt Online*, ancientegyptonline.co.uk/scribe/.

Shetty, Badreesh. "An in-Depth Guide to Supervised Machine Learning Classification." *Built In*, builtin.com/data-science/supervised-machine-learning-classification.