Program Induction Practice

Oliver Y. Hasegawa

Summary Introduction

Since I did not have any prior background in program induction, I wanted to gain an intuitive understanding of what processes it actually involves by coding it. To start, I looked for relevant materials and found the Science paper by Lake et al. (2015), in which the authors provided both data and source code. Below, I replicate and run several of their scripts to execute parts of the **Bayesian Program Learning (BPL)** process as an example of program induction. Because I currently do not have access to MATLAB, I only implemented the parts available in Python, so the full BPL pipeline was not reproduced.

What is Program Induction

According to Rule et al. (2018), first of all, inductive programming refers to the "algorithms that learn program-like structures from obsevations," which is a types of (or one of the fileds in) program synthesis. Program induction is, it seems like that, a process of concept learning which represents as programs that obtained from observed data. Now BPL seems to be a method of program induction, which I will try to replicate below. Its a generative process of a program with Bayesian methodology (Ellis et al., 2022), which authors states that "works by inferring a generative procedure represented as a symbolic program."

Exploring Program Induction with Bayesian Program Learning

Thanks to one of the authors, Brenden Lake has made materials for program induction publicly available on GitHub: https://github.com/brendenlake/omniglot

In this exercise, I explored how the BPL framework implements program induction by running part of the process in Python. For this, I referred to the Python reimplementation provided by rfeinman/pyBPL, which reproduces key components of the original BPL model described in Lake et al. (2015).

Two steps involves below: 1. Creating **Omniglot** 2. Running parts of the **BPL**

1. Creating Omniglot

Here, following omniglot repository, I have replicated a visualization process using the dataset.

The Omniglot dataset provides both images and strokes data for handwritten characters. By running codes shown below, each character is visualized as composition of multiple strokes.

It seems like this process is for visualizing compositional and generative nature of how each strokes and each characters are produced.

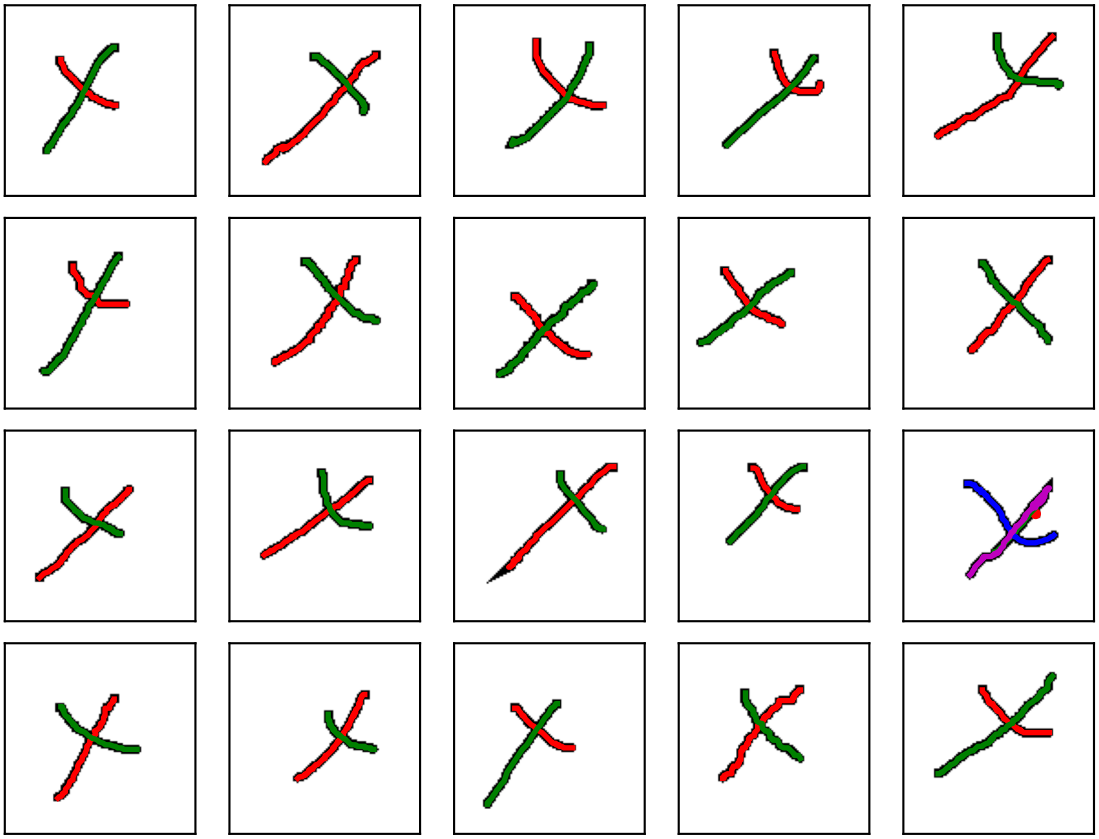
```
import numpy as np
import os
import random
from sys import platform as sys_pf
import matplotlib
if sys_pf == 'darwin':
    matplotlib.use("TkAgg")
from matplotlib import pyplot as plt

def plot_motor_to_image(I,drawing,lw=2):
    drawing = [d[:, 0:2] for d in drawing]
    drawing = [space_motor_to_img(d) for d in drawing]
    plt.imshow(I,cmap='gray')
    ns = len(drawing)
    for sid in range(ns):
        plot_traj(drawing[sid],get_color(sid),lw)
    plt.xticks([])
    plt.yticks([])

def plot_traj(stk,color,lw):
    n = stk.shape[0]
    if n > 1:
        plt.plot(stk[:,0],stk[:,1],color=color,linewidth=lw)
    else:
        plt.plot(stk[0,0],stk[0,1],color=color,linewidth=lw,marker='.')

def get_color(k):
    scol = ['r','g','b','m','c']
    ncol = len(scol)
    if k < ncol:
        out = scol[k]
    else:
        out = scol[-1]
```

```
    return out

def num2str(idx):
    if idx < 10:
        return '0'+str(idx)
    return str(idx)

def load_img(fn):
    I = plt.imread(fn)
    I = np.array(I,dtype=bool)
    return I

def load_motor(fn):
    motor = []
    with open(fn,'r') as fid:
        lines = fid.readlines()
    lines = [l.strip() for l in lines]
    for myline in lines:
        if myline =='START':
            stk = []
        elif myline =='BREAK':
            stk = np.array(stk)
            motor.append(stk)
            stk = []
        else:
            arr = np.fromstring(myline,dtype=float,sep=',')
            stk.append(arr)
    return motor

def space_motor_to_img(pt):
    pt[:,1] = -pt[:,1]
    return pt
def space_img_to_motor(pt):
    pt[:,1] = -pt[:,1]
    return pt

if __name__ == "__main__":
    img_dir = 'images_background'
    stroke_dir = 'strokes_background'
    nreps = 20
    nalpha = 5
```

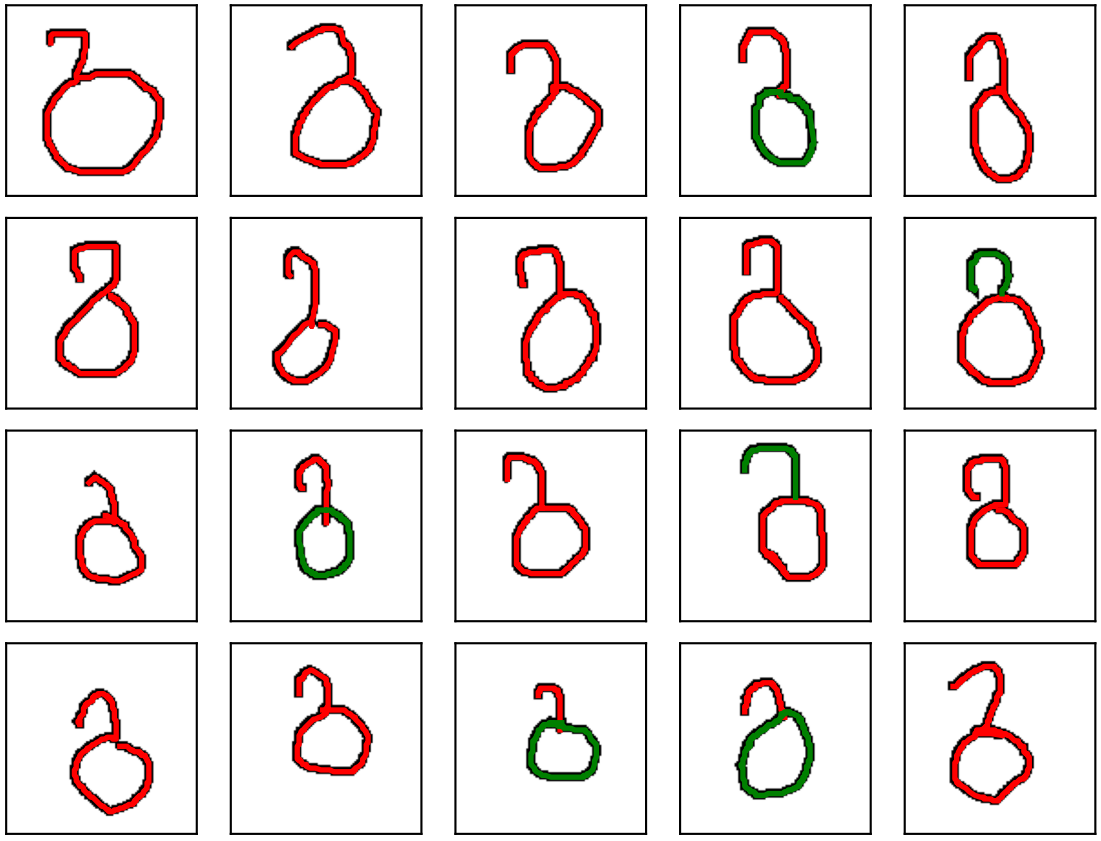
```
    alphabet_names = [a for a in os.listdir(img_dir) if a[0] != '.']
    alphabet_names = random.sample(alphabet_names,nalpha)

    for a in range(nalpha):
        print('generating figure ' + str(a+1) + ' of ' + str(nalpha))
        alpha_name = alphabet_names[a]

        character_id = random.randint(1,len(os.listdir(os.path.join(img_dir,alpha_name))))

        img_char_dir = os.path.join(img_dir,alpha_name,'character'+num2str(character_id))
        stroke_char_dir = os.path.join(stroke_dir,alpha_name,'character'+num2str(character_id

        fn_example = os.listdir(img_char_dir)[0]
        fn_base = fn_example[:fn_example.find('_')]

        plt.figure(a,figsize=(6,5))
        plt.clf()
        for r in range(1,nreps+1):
            plt.subplot(4,5,r)
            fn_stk = stroke_char_dir + '/' + fn_base + '_' + num2str(r) + '.txt'
            fn_img = img_char_dir + '/' + fn_base + '_' + num2str(r) + '.png'
            motor = load_motor(fn_stk)
            I = load_img(fn_img)
            plot_motor_to_image(I,motor)
            if r==1:
                plt.title(alpha_name[:15] + '\n character ' + str(character_id))
        plt.tight_layout()
    plt.show()
```

```
generating figure 1 of 5
generating figure 2 of 5
generating figure 3 of 5
generating figure 4 of 5
generating figure 5 of 5
```

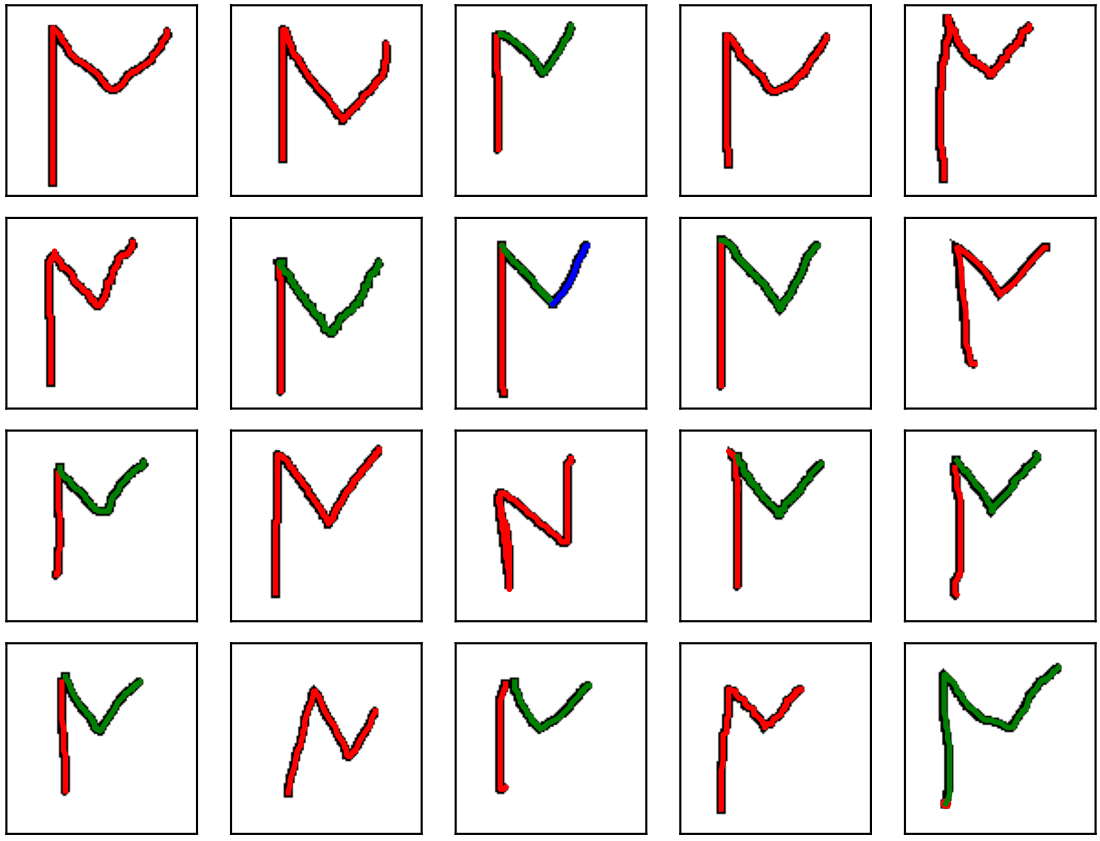
Early_Aramaic
character 22



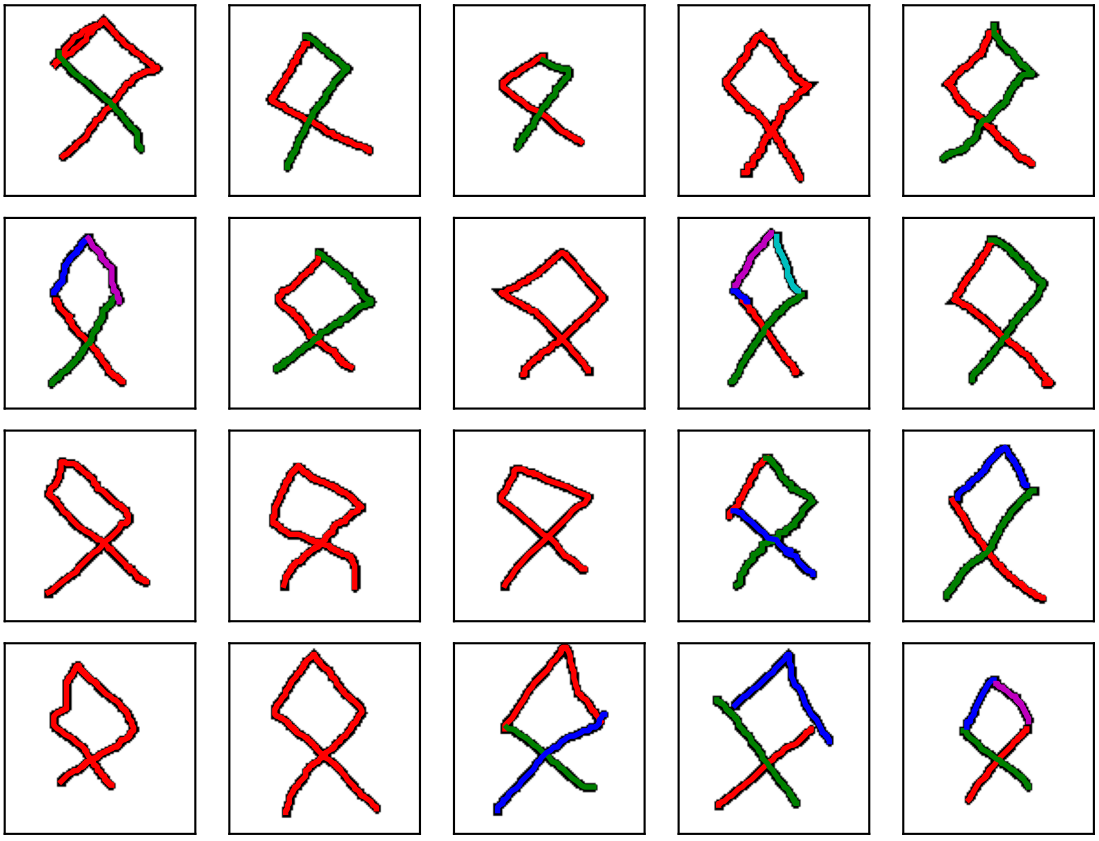
Mkhedruli_(Geor
character 3

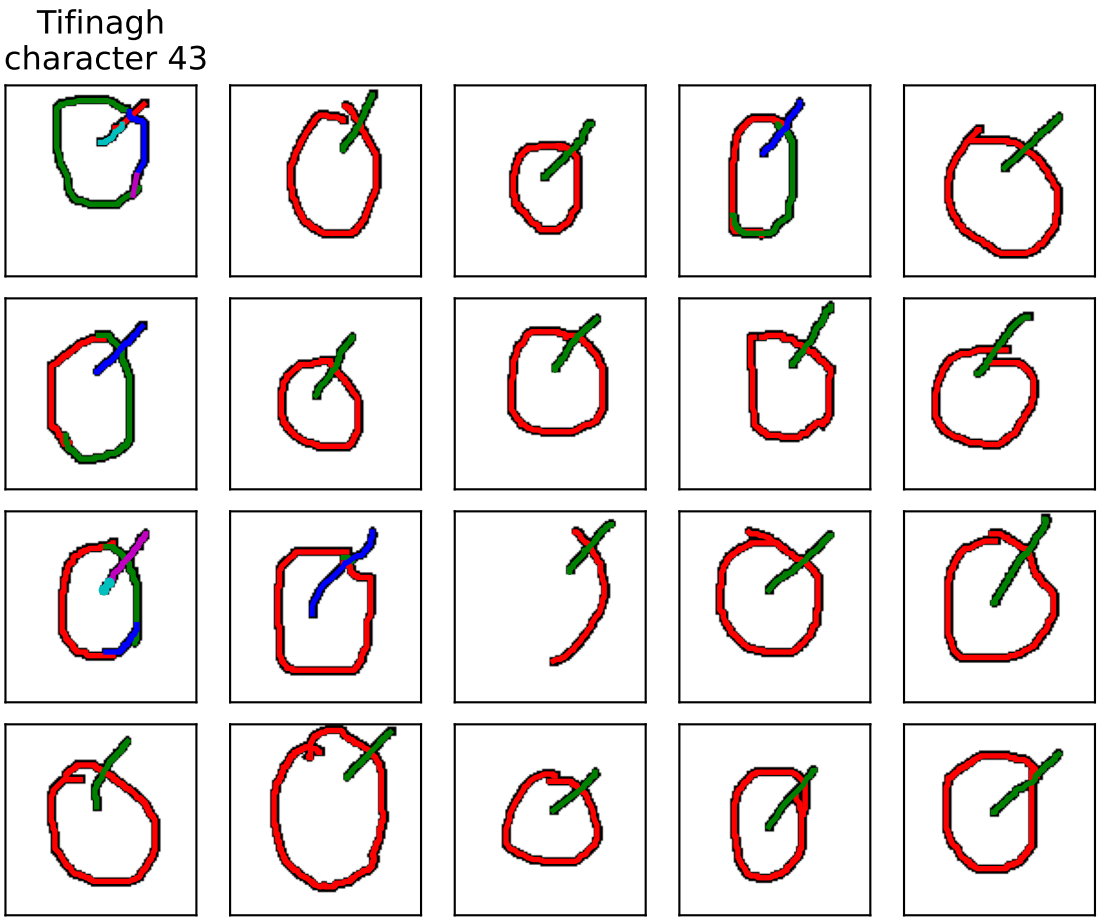


Blackfoot_(Cana
character 3



Anglo-Saxon_Fut
character 23





2. generating BPL

Now moving on, though only partially, to the actual BPL, i.e., program induction!

In implemeting this, I used the pyBPL library, a python version of the original MATLAB BPL model provided by Reuben Feinman.

Probabilistic program of handwritten character is provided through this BPL. Below code loads pre-learned (on MATLAB) parameters, samples new character and "token," then compute the log-likelihoods which is the likelihood that could potentially be used for posterior inference (something that is not covered by pyBPL).

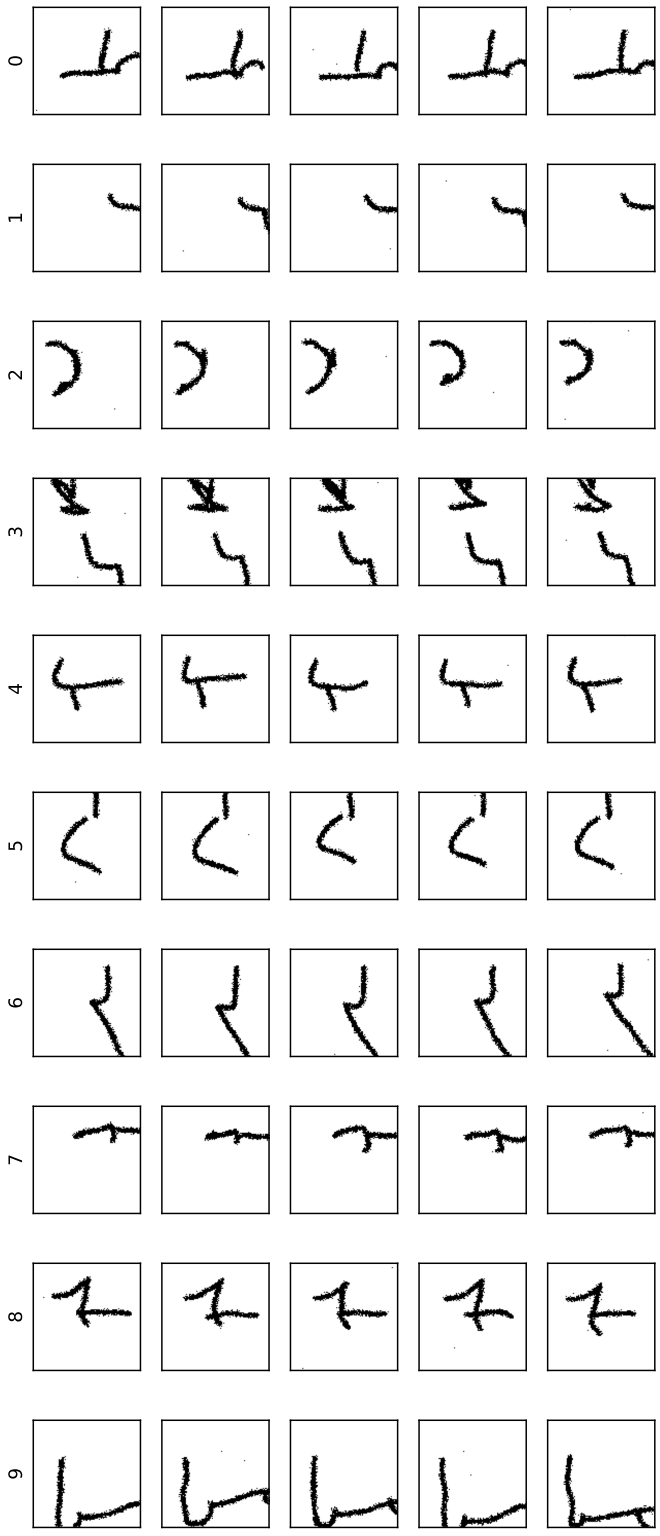
```
import matplotlib.pyplot as plt

from pyBPL.library.library import Library
```

```
from pyBPL.model.model import CharacterModel



def display_type(c):
    print("----BEGIN CHARACTER TYPE INFO----")
    print("num strokes: %i" % c.k)
    for i in range(c.k):
        print("Stroke #%i:" % i)
        print("\tsub-stroke ids: ", list(c.part_types[i].ids.numpy()))
        print("\trelation category: %s" % c.relation_types[i].category)
    print("----END CHARACTER TYPE INFO----")


def main():
    print("generating character...")
    lib = Library(use_hist=True)
    model = CharacterModel(lib)
    fig, axes = plt.subplots(nrows=10, ncols=5, figsize=(6, 15))
    for i in range(10):
        ctype = model.sample_type()
        ll = model.score_type(ctype)
        print("type %i" % i)
        display_type(ctype)
        print("log-likelihood: %0.2f \n" % ll.item())
        for j in range(5):
            ctoken = model.sample_token(ctype)
            img = model.sample_image(ctoken)
            axes[i, j].imshow(img, cmap="Greys")
            axes[i, j].tick_params(
                which="both",
                bottom=False,
                left=False,
                labelbottom=False,
                labelleft=False,
            )
        axes[i, 0].set_ylabel("%i" % i, fontsize=10)
    plt.show()


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

```
generating character...
type 0
----BEGIN CHARACTER TYPE INFO----
num strokes: 2
Stroke #0:
    sub-stroke ids:  [14, 1086]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [52]
    relation category: unihist
----END CHARACTER TYPE INFO----
log-likelihood: -96.27

type 1
----BEGIN CHARACTER TYPE INFO----
num strokes: 2
Stroke #0:
    sub-stroke ids:  [729, 715, 130, 257, 850, 53, 658, 196]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [7]
    relation category: mid
----END CHARACTER TYPE INFO----
log-likelihood: -294.27

type 2
----BEGIN CHARACTER TYPE INFO----
num strokes: 3
Stroke #0:
    sub-stroke ids:  [0]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [15]
    relation category: unihist
Stroke #2:
    sub-stroke ids:  [875, 1100]
    relation category: mid
----END CHARACTER TYPE INFO----
log-likelihood: -130.71

type 3
----BEGIN CHARACTER TYPE INFO----
num strokes: 3
```

```
Stroke #0:
    sub-stroke ids:  [44]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [8, 968, 987, 17]
    relation category: mid
Stroke #2:
    sub-stroke ids:  [375, 329, 120]
    relation category: unihist
----END CHARACTER TYPE INFO----
log-likelihood: -253.07

type 4
----BEGIN CHARACTER TYPE INFO----
num strokes: 2
Stroke #0:
    sub-stroke ids:  [819]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [45]
    relation category: mid
----END CHARACTER TYPE INFO----
log-likelihood: -69.59

type 5
----BEGIN CHARACTER TYPE INFO----
num strokes: 2
Stroke #0:
    sub-stroke ids:  [474]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [31, 11]
    relation category: unihist
----END CHARACTER TYPE INFO----
log-likelihood: -109.25

type 6
----BEGIN CHARACTER TYPE INFO----
num strokes: 1
Stroke #0:
    sub-stroke ids:  [450, 765]
    relation category: unihist
```

```
----END CHARACTER TYPE INFO----
log-likelihood: -74.56

type 7
----BEGIN CHARACTER TYPE INFO----
num strokes: 2
Stroke #0:
    sub-stroke ids:  [756, 1153]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [6]
    relation category: mid
----END CHARACTER TYPE INFO----
log-likelihood: -104.29

type 8
----BEGIN CHARACTER TYPE INFO----
num strokes: 2
Stroke #0:
    sub-stroke ids:  [547, 686]
    relation category: unihist
Stroke #1:
    sub-stroke ids:  [13]
    relation category: unihist
----END CHARACTER TYPE INFO----
log-likelihood: -95.08

type 9
----BEGIN CHARACTER TYPE INFO----
num strokes: 1
Stroke #0:
    sub-stroke ids:  [12, 440, 493, 206, 966, 653]
    relation category: unihist
----END CHARACTER TYPE INFO----
log-likelihood: -184.04
```


0
1
2
3
4
5
6
7
8
9

Log-likelihood of each generative model is provided with the figure.

Through this coding replication exercise, I was able to grasp how observed data (e.g., handwritten characters, language, objects) can be represented as structured, compositional programs. Visualizing these process helped me appreciate how compositionality function in human cognition. I am interested in how compositional knowledge structes can be entailed into program induction frameworks!

References

- Ellis, K., Albright, A., Solar-Lezama, A., Tenenbaum, J. B., & O'Donnell, T. J. (2022). Synthesizing theories of human language with Bayesian program induction. *Nature Communications, 13*(1), 5024. https://doi.org/10.1038/s41467-022-32012-w
- Lake, B. M., Salakhutdinov, R., & Tenenbaum, J. B. (2015). Human-level concept learning through probabilistic program induction. *Science, 350*(6266), 1332–1338. https://doi.org/10.1126/science.aab3050
- Rule, J., Schulz, E., Piantadosi, S. T., & Tenenbaum, J. B. (2018). Learning list concepts through program induction. In bioRxiv. *bioRxiv*. https://doi.org/10.1101/321505