ECE655 Project 05

Author: Stewart Schuler

Due: 11/16/2025

Contents

1 Part A

The pre-augmented dataset used for this project consists of 40 hand drawn 20x20 pixel *Q*, *M*, *X*, and *Other*, 10 of each class. They were generated using the paint.NET tool and saved as PNG files. The letters are shown in Figures 1- 4.

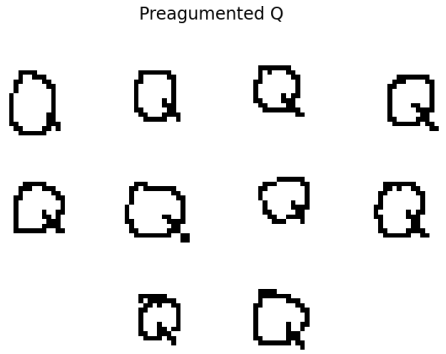

Figure 1: Q Dataset

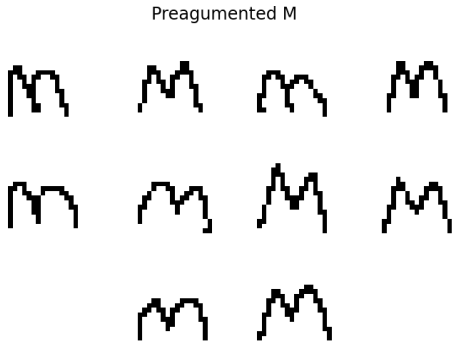

Figure 2: M Dataset

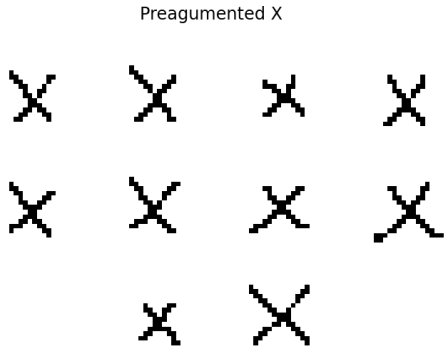

Figure 3: X Dataset

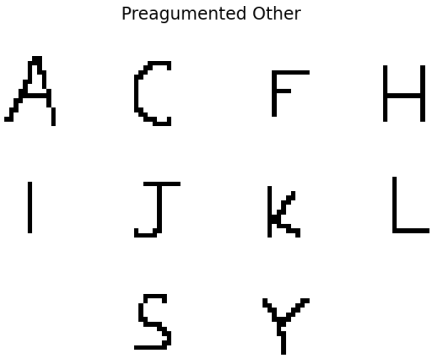

Figure 4: Other Dataset

2 Part B/C

Next, before training a model since 40 data samples is insufficient the dataset was upsampled by a factor of 10 to now contain 400 images. Originally these 400 images were copies of the 40 hand drawn images from Part A. To augment the dataset random rotations and scaling were applied to these images. To create 400 unique images, albeit they are linear transforms applied to the original images. The dataset augmentation was acheived by the following code, and the 400 post augmented images can be see in Figures 5- 8.

```
# %% Agument Dataset
composer = Compose( [RandomAffine(degrees=(-30,30), translate
    =(0.15,0.15)) ]);
images_aug = torch.from_numpy(np.empty((400,1,20,20), dtype=
    np.float32))
for ii in range(images.shape[0]):
    images_aug[ii] = composer(images[ii]);
```

Listing 1: part_b.py

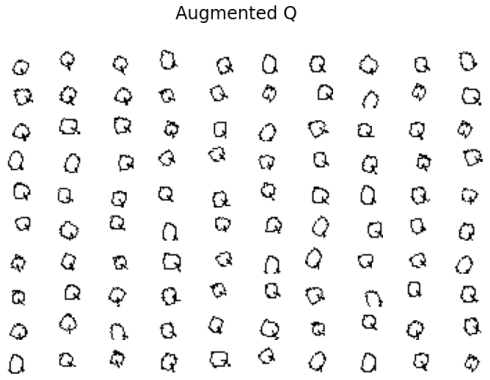

Figure 5: Augmented Q Dataset

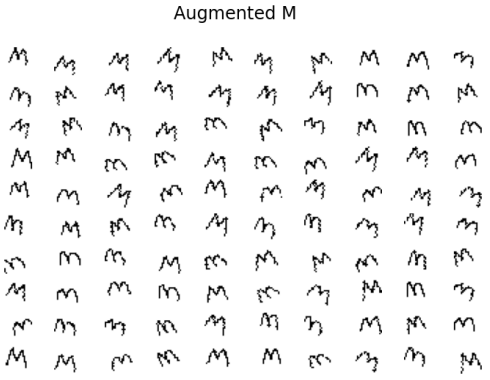

Figure 6: Augmented M Dataset

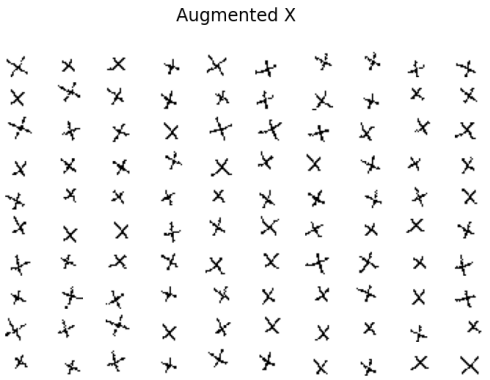

Figure 7: Augmented X Dataset

Figure 8: Augmented Other Dataset

3 Part E

Next, in order to train a model the augmented dataset an 80:20 split was applied to the data.

```
# %% Normalize the dataset
normalizer = Normalize(mean=(127.5,), std=(127.5,))
images_aug_norm = normalizer(images_aug);

# %% Randomly shuffle and partition into train and test
r_idx = np.random.permutation(400);

x = images_aug_norm[r_idx];
y = labels[r_idx];

x_train = x[  0:320:1];
x_test  = x[320:400:1];
y_train = y[  0:320:1];
y_test  = y[320:400:1];
```

Listing 2: part_d1.py

The goal of this exercise was to find a *cheap* model capable of achieving at least a 70% validation accuracy. After some experimentation the model able to achieve the stated goal was a single convolution layer with a 5x5 kernel, a max pool layer which decreases the kernel output by a factor of 16, and a linear output layer with 4 neurons for each class. In total the model used only 94 parameters to achieve a best validation accuracy of 0.75%.

```
model = nn.Sequential(
    nn.Conv2d(1, 1, kernel_size=5, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(4),
    nn.Flatten(),
    nn.Linear(16, 4)
).to(device);
```

Listing 3: part_d2.py

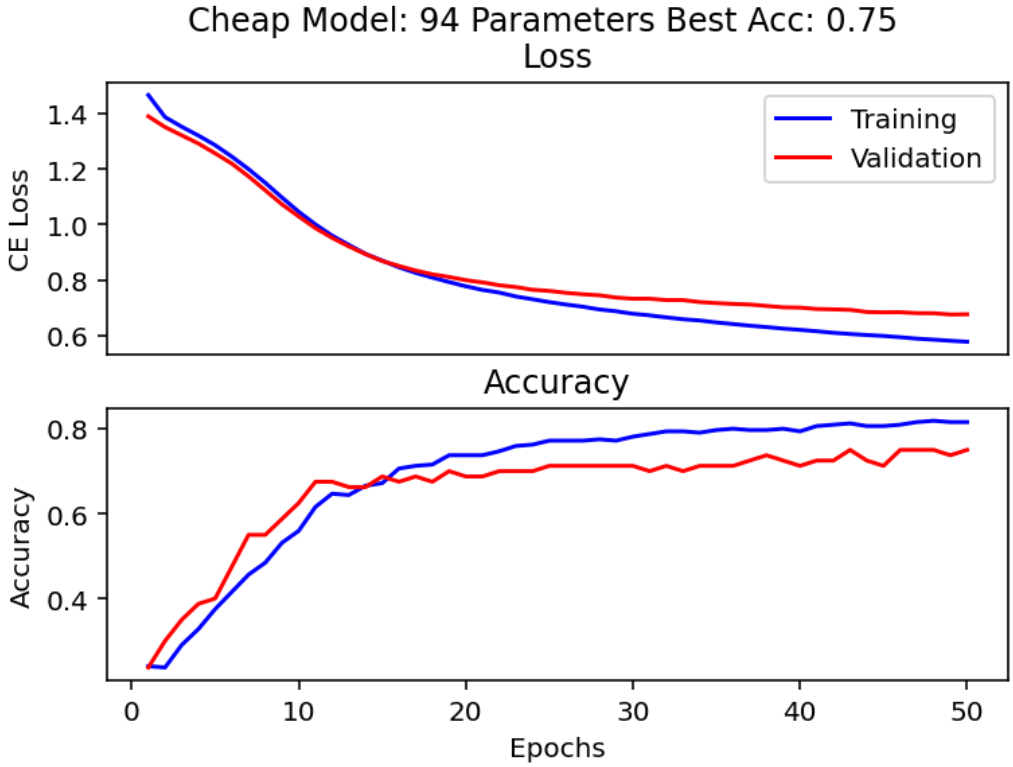

Figure 9: Cheap Model Loss and Accuracy

4 Part F

Next, we consider a range of different models. For this section 60 models were chosen. The models are defined by randomly selecting 60 unique permutations of the following parameter sets.

```
# %% Create 60 random model definitions
num_features = [2,4,6,8,10,12];
kernel_size = [3,5];
kernel_depth = [1,2];
num_hidden = [0,1,2];
hidden_size = [64,128,256,512];

all_combinations = list(itertools.product(
    num_features,
    kernel_size,
    kernel_depth,
    num_hidden,
    hidden_size
))
random_sets = random.sample(all_combinations, 60)
```

Listing 4: part_f1.py

The models are then constructed based on those 5 parameters using the following *create_model* function.

```
def get_output_size( model ):
    with torch.no_grad():
        dummy = torch.zeros(1, 1, 20, 20)
        out = model(dummy)
        flat_size = out.shape[1]
    return flat_size;

def create_model( num_features, kernel_size, kernel_depth,
   num_hidden, hidden_size ):
    model = nn.Sequential();
    for ii in range(kernel_depth):
        model.add_module(f"conv{ii}", nn.Conv2d(
   get_output_size(model), num_features, kernel_size, padding
   =0));
        model.add_module(f"ReLU{ii}", nn.ReLU());
        model.add_module(f"maxp{ii}", nn.MaxPool2d(2));

    model.add_module("flatten", nn.Flatten());
    for ii in range(num_hidden):
        model.add_module(f"h{ii}", nn.Linear( get_output_size
   (model), int(hidden_size)));
```

```
18          model.add_module(f"hReLU{ii}", nn.ReLU());
19      model.add_module("output", nn.Linear(get_output_size(
        model),4));
20      model = model.to(device)
21      return model;
```

Listing 5: part_f2.py

The same 80:20 data split from the previous section was then used to train and validate all 60 models. The results of which are shown in the table in Section 6. All the models that were able to reach high accuracy (i.e. E95 is not *None*), had at least 6 convolution kernels, with most of them having 10 or 12. Conversely none of the models with 2 or 4 kernels were able to achieve a 95% validation accuracy, which make sense if we assume with 4+ kernels, the kernels are learning features to identify a specific class. As for the cost the parameters that seemed to have the largest impact were the number of hidden layers and the number of neurons per hidden layer. Which in turn ballooned the number of parameters in the total model. Because the convolution kernels were fairly small in size, increasing the the number of kernels didn't have that large an impact on number of parameters.

The best performing model was model 21. Which had the following loss and accuracy curves, and it produced a validation test accuracy of 96.25%, shown by Figure 11. And it can be seen that one of the validation set data points it got incorrect, was heavily effected by the augmentation making it a very difficult image to classify. Also interesting the best performing model used 17496 parameters, while significant more than the cheap model, it was substantially less than some of the other worse performing models.

The model architecture of the best performing model is as follows,

```
1  Sequential(
2    (conv0): Conv2d(1, 12, kernel_size=(5, 5), stride=(1, 1))
3    (ReLU0): ReLU()
4    (maxp0): MaxPool2d(kernel_size=2, stride=2, padding=0,
       dilation=1, ceil_mode=False)
5    (conv1): Conv2d(12, 12, kernel_size=(5, 5), stride=(1, 1))
6    (ReLU1): ReLU()
7    (maxp1): MaxPool2d(kernel_size=2, stride=2, padding=0,
       dilation=1, ceil_mode=False)
8    (flatten): Flatten(start_dim=1, end_dim=-1)
9    (h0): Linear(in_features=48, out_features=256, bias=True)
10   (hReLU0): ReLU()
11   (output): Linear(in_features=256, out_features=4, bias=True
       )
12 )
```

Listing 6: part_f3.py

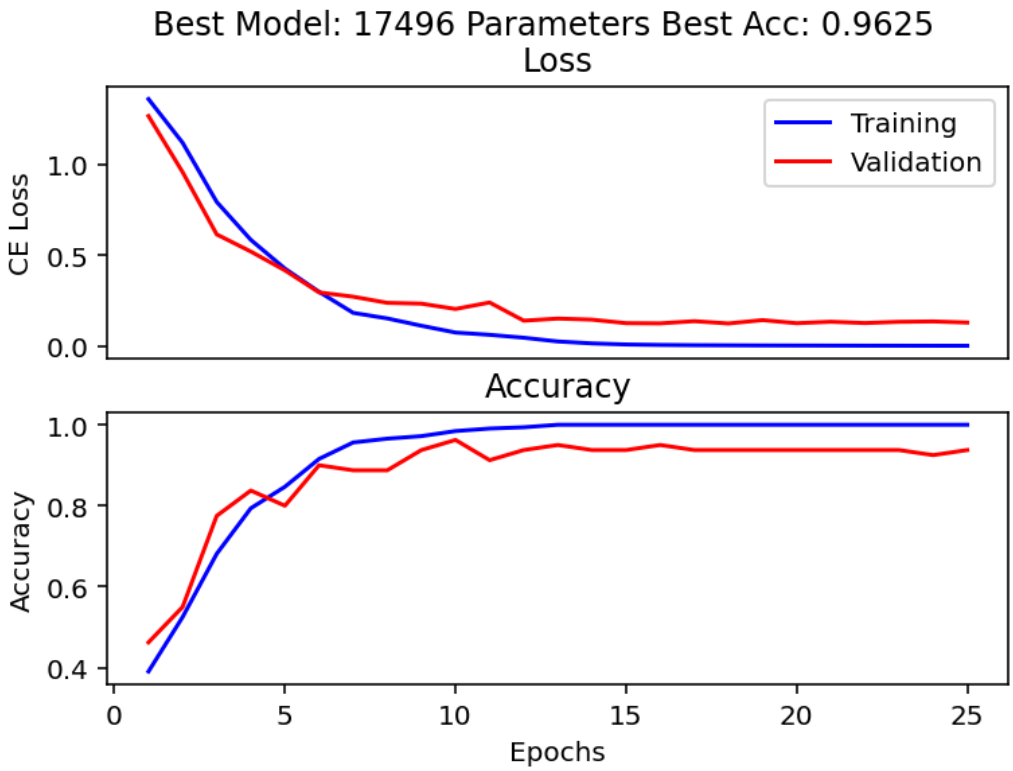

Figure 10: Best Model Loss and Accuracy

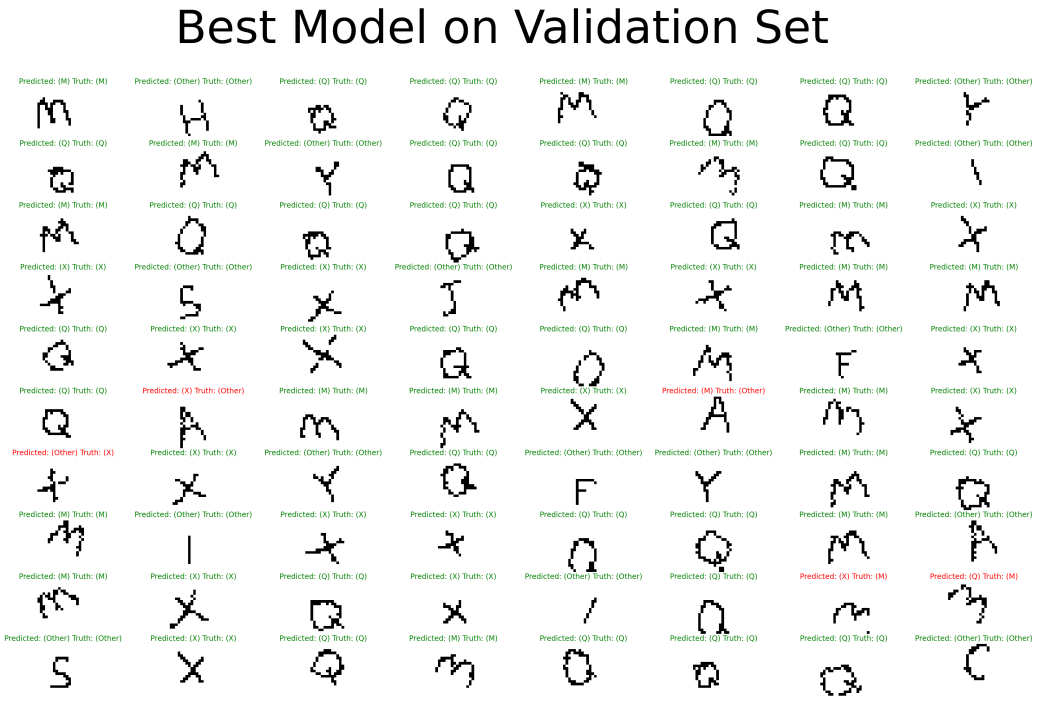

Figure 11: Best Model Validation Results

5 Part X

Finally, we consider how the models perform on a distinctly generated test set. Per the assignment instructions the test set was generated as a single .PNG image as shown in Figure 12. The easy other images are newly drawn versions of the letters found in the other training set. For the tricky letters a chose a set of greek letters that slightly resemble those the classes Q, M, and X. The explanation for each tricky character is as follows;

α: has a round portion like Q while also having a cross like portion of the X.
Ω: a lot like a Q but isn't fully closed.
λ: a lot like an X but is missing an arm.
Θ: is a Q with the tail shifted to the center of the circle.
β: has the same arches as a sideways M while also having a Q like tail.

Using the following code each individual image was extracted using the *PIL* API. And the individually extracted images are shown in the format used previously in Figure 13.

```
test_tensor = torch.empty((25,1,20,20));
```

```
tt = ToTensor();
grid_img = Image.open("../dataset/test_set.png");
grid_img.show()
# Select subimages and convert from PIL to tensor
crop_bounds = np.array([2,2,22,22]);
for col_idx in range(5):
    crop_bounds[0] = 2;
    crop_bounds[2] = 22;
    for row_idx in range(5):
        test_tensor[row_idx+(5*col_idx)] = tt(grid_img.crop(
   crop_bounds))[0]
        crop_bounds[0] += 22;
        crop_bounds[2] += 22;
    crop_bounds[1] += 22;
    crop_bounds[3] += 22;
```

Listing 7: part_x.py

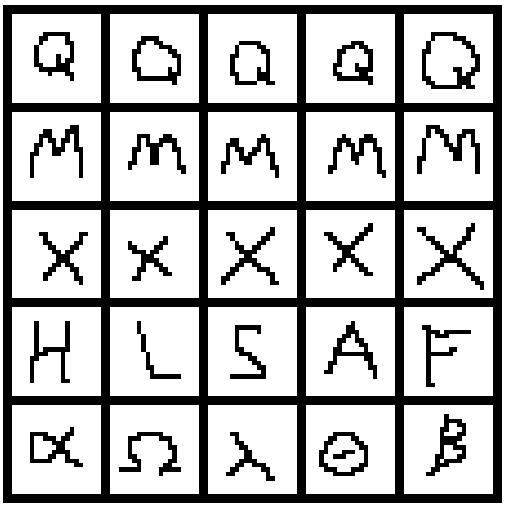

Figure 12: Test Set Image

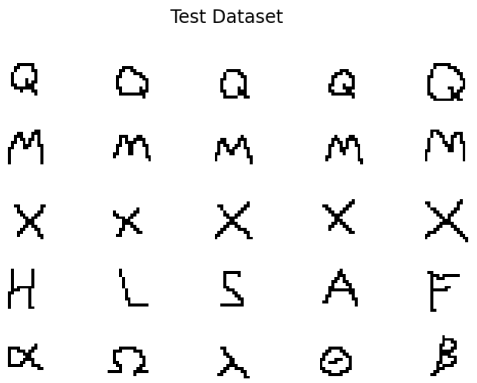

Figure 13: Test Set Extracted

We test this new dataset on our best performing model (21) and produced the predictions shown in Figure 14. As can be seen, the model correctly predicts all Q, M, and X test cases, and got 4/5 of the *easy* other classes, struggling to identify the A case. 19/20 of the expected models is about in line what we should expect for performance given the previously reported 96.25% validation accuracy. This model however failed all *tricky* cases in a predictable manor given it predicted what we were attempting to trick it with, excepting the β which it predicted X for.

We repeat the same test with our worse performing model (28) and produces the predictions shown in Figure 15. It can be seen that the model

Figure 14: Best Model on Test Set

completely failed on the Other class. This leads me to conclude that the model completely ignored learning the other class, likely due to the diversity of characters and instead focused on learning filters geared towards identifying the three consistent classes. Also this model architecture only had 4 convolution filters, which means at most it is identifying one features per class.

Figure 15: Worst Model on Test Set

6 Appendix A

Model Num	min BC Loss	E95	E90	E80	E70	Parameters	Eval Time (Sec)	Cost
0	0.31720	None	16	5	3	6556	0.000269	1.761770
1	0.29820	None	12	4	1	66924	0.000179	11.973964
2	0.21577	None	6	2	2	99260	0.000181	17.993062
3	0.40273	None	11	4	2	96152	0.000297	28.557183
4	0.22050	None	9	3	2	334420	0.000180	60.129843
5	0.38746	None	None	7	5	956	0.000202	0.193522
6	0.29039	None	8	5	4	29686	0.000297	8.827017
7	0.22730	None	11	4	3	25260	0.000224	5.653819
8	0.26861	None	11	2	1	396092	0.000180	71.304613
9	0.37929	None	23	14	9	576	0.000200	0.115377
10	0.40754	None	None	10	5	7946	0.000249	1.982340
11	0.37884	None	23	4	2	52264	0.000196	10.235226
12	0.35326	None	None	6	2	2260	0.000135	0.304329
13	0.40014	None	None	6	3	34104	0.000182	6.208934
14	0.18554	8	6	3	1	133740	0.000180	24.087079
15	0.26568	None	7	3	2	62912	0.000180	11.332189
16	0.18335	16	12	8	3	4120	0.000202	0.830453
17	0.30250	None	23	6	3	25334	0.000253	6.411139
18	0.32923	None	6	4	2	264916	0.000181	47.975754
19	0.46278	None	None	9	3	8568	0.000181	1.549421
20	0.36908	None	None	4	4	25056	0.000179	4.474806
21	0.12432	10	6	4	3	17496	0.000252	4.409473
22	0.42005	None	None	7	5	251456	0.000183	46.081937
23	0.28672	None	14	7	4	1162	0.000204	0.236586
24	0.24784	None	14	10	5	4120	0.000205	0.844008
25	0.20833	10	6	4	2	25814	0.000252	6.493037
26	0.34001	None	23	13	10	1162	0.000204	0.236844
27	0.24983	19	10	6	5	11292	0.000249	2.811219
28	0.64264	None	None	None	14	336	0.000204	0.068391
29	0.53851	None	None	8	4	76480	0.000297	22.697451
30	0.35125	None	None	7	3	16812	0.000178	2.999085
31	0.32205	None	None	6	2	2676	0.000135	0.360073
32	0.28077	None	10	4	4	125180	0.000183	22.955465
33	0.26102	None	11	5	2	1696	0.000136	0.229950
34	0.20411	25	16	6	2	2008	0.000134	0.269193
35	0.29356	None	14	8	6	2934	0.000205	0.601140
36	0.23121	15	7	4	3	23068	0.000298	6.868530
37	0.76458	None	None	None	11	5694	0.000298	1.697419
38	0.13756	None	4	4	3	330808	0.000228	75.578707
39	0.22298	None	7	3	2	33300	0.000182	6.066766
40	0.33855	None	None	5	5	12824	0.000296	3.794761
41	0.14698	16	8	5	5	680040	0.000228	155.086198
42	0.50541	None	None	13	4	71742	0.000335	24.067064
43	0.30691	None	None	5	4	30360	0.000252	7.653003
44	0.38848	None	19	4	2	191552	0.000229	43.890726
45	0.35905	None	None	7	4	233044	0.000230	53.574566
46	0.38958	None	None	12	7	4170	0.000263	1.097623
47	0.17565	11	4	3	2	165536	0.000232	38.334554
48	0.56923	None	None	5	2	79424	0.000225	17.885011
49	0.22258	None	5	3	1	250236	0.000187	46.680487
50	0.21474	18	6	4	3	208744	0.000183	38.157967
51	0.36164	None	None	7	4	1696	0.000135	0.228309
52	0.27809	None	8	4	3	74282	0.000296	21.962076
53	0.52903	None	None	17	8	11838	0.000252	2.977490
54	0.62445	None	None	None	5	672	0.000133	0.089445
55	0.25641	None	19	6	3	3344	0.000134	0.447572
56	0.31852	None	16	8	5	5440	0.000250	1.357476
57	0.29765	None	10	3	1	99744	0.000180	17.973070
58	0.45208	None	None	16	12	610	0.000203	0.123802
59	0.21955	None	4	3	2	417384	0.000182	75.793391

Table 1: Model Results