Exercise- Looking at the outcome of a multi-class classifier

We are going to load up the MNIST digit data again

-make sure it is converted to numpy, depending on your version of scikit lean, it may load as a pd frame

-split into test and train

-create a NN classifier for all ten classes

-look at the performance of the ten category classifier

Get the data

In [1]:
```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/dat
asets/_openml.py:968: FutureWarning: The default va
lue of `parser` will change from `'liac-arff'` to
`'auto'` in 1.4. You can set `parser='auto'` to sil
ence this warning. Therefore, an `ImportError` will
be raised from 1.4 if the dataset is dense and pand
as is not installed. Note that the pandas parser ma
y return different data types. See the Notes Sectio
n in fetch_openml's API doc for details.
  warn(
```

Out[1]:
```
dict_keys(['data', 'target', 'frame', 'categories',
'feature_names', 'target_names', 'DESCR', 'detail
s', 'url'])
```

check sizes

In [2]:
```
X,y=mnist["data"],mnist["target"]
print(X.shape)
print (y.shape)
```

```
(70000, 784)
(70000,)
```

convert to numpy if these are dataframes-

In [3]:
```
# you may or may not need these lines
X=X.to_numpy()
y=y.to_numpy()
```

Check to see if the conversion worked correctly or not

The last time we worked with the MNIST digits set, we did not convert from Pandas to Numpy form, but there were a couple of cases where we had to do conversions later in the process.

In this notebook I just did the conversion up front

In [4]:
```
type(X)
```

Out[4]:
```
numpy.ndarray
```

Plot just to check behavior

In [5]:
```
%matplotlib inline
```

In [6]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt

# do a reshape here to make this a 28 x 28 binary i

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```

Set up test and train, set up the model and train it

In [7]:
```
X_train, X_test, y_train, y_test = X[:60000], X[600
```

Note that we did not set up y to be a set of true/false values, but rather are using the full set of number values

When we have only two categories, this is called a binary classification. Logistic regression is the classic form of a binary classification

With more than two categories, the problem becomes a multi-category classification.

Typically we want to predict the probability that the input data is in each of the M categories we are working with. The probabilities need to be normalized so the probabilities of the M categories add up to one.

Usually, we assign the data to the category with the highest probability associated with it.

We are still using the SciKit Learn implementation of a neural net classifier. This function is set up like most other predictive models in SciKit Learn and it allows us to build basic Neural Network classifiers and predictors. We will work with this implementation of the model for a couple of weeks, then later in the semester we will work with the TensorFlow package, which offers much more control and options in working with Neural nets.

The two main "pro-level" tools for working with Neural Networks are TensorFlow and PyTorch, with Theano a third option. Once you are comfortable with TensorFlow, you can work through a book or tutorial to learn PyTorch and Theano.

In [8]:
```
from sklearn.neural_network import MLPClassifier
clf = MLPClassifier(solver='adam', alpha=1e-5, rand

clf.fit(X_train, y_train)
```

```
Iteration 1, loss = 2.75341305
Iteration 2, loss = 2.19013267
Iteration 3, loss = 1.95206692
Iteration 4, loss = 1.71859520
Iteration 5, loss = 1.54210350
Iteration 6, loss = 1.42070221
Iteration 7, loss = 1.29888508
Iteration 8, loss = 1.19309644
Iteration 9, loss = 1.01536968
Iteration 10, loss = 0.94127746
Iteration 11, loss = 0.89634172
Iteration 12, loss = 0.86530786
Iteration 13, loss = 0.83276653
Iteration 14, loss = 0.78043688
Iteration 15, loss = 0.71152714
Iteration 16, loss = 0.64162008
Iteration 17, loss = 0.58964663
Iteration 18, loss = 0.55348038
Iteration 19, loss = 0.52707966
Iteration 20, loss = 0.49977887
Iteration 21, loss = 0.47715709
Iteration 22, loss = 0.45147110
Iteration 23, loss = 0.43357121
Iteration 24, loss = 0.41474531
Iteration 25, loss = 0.38545039
Iteration 26, loss = 0.31513471
Iteration 27, loss = 0.29380048
Iteration 28, loss = 0.26866502
Iteration 29, loss = 0.25338404
Iteration 30, loss = 0.24432608
Iteration 31, loss = 0.23597142
Iteration 32, loss = 0.22769343
Iteration 33, loss = 0.21866569
Iteration 34, loss = 0.21571418
Iteration 35, loss = 0.20812231
Iteration 36, loss = 0.20284147
Iteration 37, loss = 0.20039210
Iteration 38, loss = 0.19488330
Iteration 39, loss = 0.19087340
Iteration 40, loss = 0.18680808
```

```
Iteration 41, loss = 0.18343635
Iteration 42, loss = 0.17896808
Iteration 43, loss = 0.17878711
Iteration 44, loss = 0.17652935
Iteration 45, loss = 0.17121849
Iteration 46, loss = 0.17236696
Iteration 47, loss = 0.17022000
Iteration 48, loss = 0.16492951
Iteration 49, loss = 0.16622504
Iteration 50, loss = 0.16500369
Iteration 51, loss = 0.15826132
Iteration 52, loss = 0.15941875
Iteration 53, loss = 0.15681262
Iteration 54, loss = 0.15632673
Iteration 55, loss = 0.15637187
Iteration 56, loss = 0.15148760
Iteration 57, loss = 0.15072408
Iteration 58, loss = 0.14961836
Iteration 59, loss = 0.14726600
Iteration 60, loss = 0.14531482
Iteration 61, loss = 0.14734654
Iteration 62, loss = 0.14285941
Iteration 63, loss = 0.14075832
Iteration 64, loss = 0.14177118
Iteration 65, loss = 0.14052871
Iteration 66, loss = 0.13691057
Iteration 67, loss = 0.13988414
Iteration 68, loss = 0.14178429
Iteration 69, loss = 0.13569088
Iteration 70, loss = 0.13322628
Iteration 71, loss = 0.13281257
Iteration 72, loss = 0.13104760
Iteration 73, loss = 0.13710205
Iteration 74, loss = 0.13545744
Iteration 75, loss = 0.13109232
Iteration 76, loss = 0.13140565
Iteration 77, loss = 0.13059977
Iteration 78, loss = 0.12673189
Iteration 79, loss = 0.12674905
Iteration 80, loss = 0.12487752
```

```
Iteration 81, loss = 0.12381117
Iteration 82, loss = 0.12266894
Iteration 83, loss = 0.12722033
Iteration 84, loss = 0.12317736
Iteration 85, loss = 0.12322361
Iteration 86, loss = 0.11822460
Iteration 87, loss = 0.11849020
Iteration 88, loss = 0.11969402
Iteration 89, loss = 0.11827865
Iteration 90, loss = 0.11602736
Iteration 91, loss = 0.11607830
Iteration 92, loss = 0.11556205
Iteration 93, loss = 0.11335602
Iteration 94, loss = 0.11306923
Iteration 95, loss = 0.11428512
Iteration 96, loss = 0.11531916
Iteration 97, loss = 0.11405522
Iteration 98, loss = 0.10958438
Iteration 99, loss = 0.10977775
Iteration 100, loss = 0.10748821
Iteration 101, loss = 0.10720394
Iteration 102, loss = 0.10829296
Iteration 103, loss = 0.11093297
Iteration 104, loss = 0.10874004
Iteration 105, loss = 0.10596878
Iteration 106, loss = 0.10361696
Iteration 107, loss = 0.10538387
Iteration 108, loss = 0.10313700
Iteration 109, loss = 0.10747662
Iteration 110, loss = 0.10111290
Iteration 111, loss = 0.10638399
Iteration 112, loss = 0.10281598
Iteration 113, loss = 0.09961319
Iteration 114, loss = 0.09908888
Iteration 115, loss = 0.10240098
Iteration 116, loss = 0.10304471
Iteration 117, loss = 0.10638235
Iteration 118, loss = 0.10054280
Iteration 119, loss = 0.10127552
Iteration 120, loss = 0.09825062
```

```
Iteration 121, loss = 0.10508443
Iteration 122, loss = 0.09962343
Iteration 123, loss = 0.10406291
Iteration 124, loss = 0.09529859
Iteration 125, loss = 0.09800267
Iteration 126, loss = 0.09694503
Iteration 127, loss = 0.09774495
Iteration 128, loss = 0.09765012
Iteration 129, loss = 0.09674055
Iteration 130, loss = 0.09593553
Iteration 131, loss = 0.09415997
Iteration 132, loss = 0.09293059
Iteration 133, loss = 0.09670562
Iteration 134, loss = 0.09484736
Iteration 135, loss = 0.09395376
Iteration 136, loss = 0.09550179
Iteration 137, loss = 0.09545284
Iteration 138, loss = 0.09223235
Iteration 139, loss = 0.09152753
Iteration 140, loss = 0.09136282
Iteration 141, loss = 0.09155547
Iteration 142, loss = 0.09553251
Iteration 143, loss = 0.09327550
Iteration 144, loss = 0.09268313
Iteration 145, loss = 0.09461120
Iteration 146, loss = 0.08958613
Iteration 147, loss = 0.09068715
Iteration 148, loss = 0.09058116
Iteration 149, loss = 0.08720574
Iteration 150, loss = 0.08842850
Iteration 151, loss = 0.09123843
Iteration 152, loss = 0.09068581
Iteration 153, loss = 0.08790381
Iteration 154, loss = 0.08914643
Iteration 155, loss = 0.08666970
Iteration 156, loss = 0.08680616
Iteration 157, loss = 0.09037503
Iteration 158, loss = 0.09127909
Iteration 159, loss = 0.08645431
Iteration 160, loss = 0.08694324
```

```
Iteration 161, loss = 0.08557239
Iteration 162, loss = 0.08682728
Iteration 163, loss = 0.08856257
Iteration 164, loss = 0.08635108
Iteration 165, loss = 0.08384728
Iteration 166, loss = 0.08710934
Iteration 167, loss = 0.08781470
Iteration 168, loss = 0.08346005
Iteration 169, loss = 0.08702639
Iteration 170, loss = 0.08928152
Iteration 171, loss = 0.08472884
Iteration 172, loss = 0.08547574
Iteration 173, loss = 0.08208115
Iteration 174, loss = 0.08282215
Iteration 175, loss = 0.08064504
Iteration 176, loss = 0.08358255
Iteration 177, loss = 0.08387232
Iteration 178, loss = 0.08096928
Iteration 179, loss = 0.08440165
Iteration 180, loss = 0.08395891
Iteration 181, loss = 0.08189833
Iteration 182, loss = 0.08162309
Iteration 183, loss = 0.08276111
Iteration 184, loss = 0.07650979
Iteration 185, loss = 0.07820265
Iteration 186, loss = 0.07864637
Iteration 187, loss = 0.07977110
Iteration 188, loss = 0.07921470
Iteration 189, loss = 0.07999972
Iteration 190, loss = 0.08175143
Iteration 191, loss = 0.07523821
Iteration 192, loss = 0.08042976
Iteration 193, loss = 0.07834880
Iteration 194, loss = 0.07915074
Iteration 195, loss = 0.07572555
Iteration 196, loss = 0.07674226
Iteration 197, loss = 0.07501698
Iteration 198, loss = 0.07388013
Iteration 199, loss = 0.07640514
Iteration 200, loss = 0.07845407
```

```
Iteration 201, loss = 0.07206131
Iteration 202, loss = 0.07113683
Iteration 203, loss = 0.08298496
Iteration 204, loss = 0.07469307
Iteration 205, loss = 0.07494571
Iteration 206, loss = 0.07142103
Iteration 207, loss = 0.07391176
Iteration 208, loss = 0.07046768
Iteration 209, loss = 0.07750849
Iteration 210, loss = 0.07805928
Iteration 211, loss = 0.07859545
Iteration 212, loss = 0.07437474
Iteration 213, loss = 0.07084227
Iteration 214, loss = 0.06963790
Iteration 215, loss = 0.07375793
Iteration 216, loss = 0.07093186
Iteration 217, loss = 0.07257524
Iteration 218, loss = 0.07228602
Iteration 219, loss = 0.07428402
Iteration 220, loss = 0.07159415
Iteration 221, loss = 0.07206877
Iteration 222, loss = 0.07097364
Iteration 223, loss = 0.06897317
Iteration 224, loss = 0.07149619
Iteration 225, loss = 0.07160090
Iteration 226, loss = 0.07272293
Iteration 227, loss = 0.06832136
Iteration 228, loss = 0.06949127
Iteration 229, loss = 0.07275795
Iteration 230, loss = 0.06669590
Iteration 231, loss = 0.06578823
Iteration 232, loss = 0.07292907
Iteration 233, loss = 0.06818168
Iteration 234, loss = 0.07071788
Iteration 235, loss = 0.07073439
Iteration 236, loss = 0.06589454
Iteration 237, loss = 0.07075730
Iteration 238, loss = 0.06731947
Iteration 239, loss = 0.06788265
Iteration 240, loss = 0.06756677
```

```
Iteration 241, loss = 0.06915443
Iteration 242, loss = 0.07583090
Training loss did not improve more than tol=0.00010
0 for 10 consecutive epochs. Stopping.
```

Out[8]:

```
▾ MLPClassifier
MLPClassifier(alpha=1e-05, hidden_layer_size
s=(20, 10, 5), max_iter=500,
              random_state=1, verbose=True)
```

In [9]:

```
# prediction outputs

# This model gives us two outputs,  the fundamental

# In the cell below, for the first specimen,  the p

# This is a clear claim that the object is a 7,  as
```

```
  File "<ipython-input-9-8fb0abc82168>", line 3
    This model gives us two outputs,  the fundament
al output is actually a probability output for each
class for each specimen
         ^
SyntaxError: invalid syntax
```

In [11]:

```
clf.predict_proba(X_test[:10,:])
```

Out[11]:
```
array([[3.05501458e-076, 4.04183830e-009, 1.2964469
1e-014,
         4.93930723e-008, 6.66231085e-012, 4.4189227
3e-024,
         1.32718214e-201, 9.99999946e-001, 2.9179948
1e-053,
         2.20153016e-010],
        [2.18592006e-074, 3.85351012e-024, 1.0000000
0e+000,
         3.41184793e-010, 1.40795869e-036, 7.4881297
4e-044,
         3.29760094e-095, 2.66392321e-011, 5.1104869
8e-050,
         4.12649051e-057],
        [1.19692611e-021, 9.99994134e-001, 2.1741649
5e-006,
         3.23678015e-006, 6.20155059e-013, 4.1307385
9e-008,
         6.06823442e-019, 3.97725434e-007, 3.1580109
8e-010,
         1.60677469e-008],
        [9.99463166e-001, 2.97589445e-012, 7.6674333
3e-005,
         1.32770102e-008, 4.38726855e-004, 2.1010165
9e-007,
         1.96319887e-005, 7.20387148e-009, 1.5699460
8e-006,
         7.89012219e-012],
        [3.53611739e-041, 5.45408324e-018, 1.5614780
8e-017,
         5.13385811e-013, 9.99941171e-001, 7.9621381
9e-017,
         2.76817489e-158, 1.22838840e-005, 7.3272617
6e-040,
         4.65449216e-005],
        [5.34360702e-036, 9.99999993e-001, 3.5853988
8e-010,
         6.64113053e-009, 4.67901426e-021, 1.3926781
3e-013,
         1.71034956e-031, 4.41906622e-010, 4.3552743
```

```
0e-017,
         5.05813290e-013],
        [8.39194537e-031, 1.54197285e-013, 3.1759025
4e-013,
         3.22286005e-010, 9.99578761e-001, 1.6699604
8e-012,
         3.56371329e-118, 7.63407594e-005, 1.0302581
0e-029,
         3.44898164e-004],
        [1.46462855e-063, 5.96339080e-008, 4.2750626
0e-022,
         1.32386901e-004, 4.46840900e-012, 7.7493838
8e-017,
         3.59455828e-190, 1.51034728e-004, 7.2041301
7e-036,
         9.99716519e-001],
        [1.93521873e-005, 3.48794397e-004, 1.6068956
4e-003,
         2.67787610e-003, 1.02755609e-004, 9.8225147
4e-001,
         9.73872544e-003, 2.75261420e-008, 2.6889878
9e-003,
         5.65110890e-004],
        [7.11342396e-073, 1.20312382e-011, 8.0380840
9e-022,
         1.33708681e-008, 3.34397424e-006, 1.2075102
5e-020,
         1.16332309e-233, 1.71319742e-001, 4.0274598
2e-052,
         8.28676901e-001]])
```

The predict function of clf converts the lists of probabilities to a single numerical target

In [12]:
```
clf.predict(X_test[:10,:])
```

Out[12]:
```
array(['7', '2', '1', '0', '4', '1', '4', '9', '5',
'9'], dtype='<U1')
```

We will be looking at a number of different types of classifiers during the course, virtually all of them produce this type of output, both a probability and a category

Many of the different methods in SciKit Learn all have the same types of functions, with similar data inputs, performance measures and outputs, so it is easy to use different models in a project, or to start using new models.

We can compare these to the true classifications

In [13]:
```
y_test[0:10]
```

Out[13]:
```
array(['7', '2', '1', '0', '4', '1', '4', '9', '5',
'9'], dtype=object)
```

Lets look to see how well the predictions work on the whole set of training data

This is called a "resubstitution rate" of classification, we are looking at the prediction success on the same data that we used to train the model with. This almost always overstates the model performance due to overfitting

Last time, we loaded as using cross validation rates of classification, we could do that here as, and probably should, but it is slow with NN models

At the end of this workbook, we will look at the performance of the classifier on the test data

In [14]:
```
y_pred=clf.predict(X_train)
```

For me, the confusion matrix is really the key to understanding the classification performance

In [15]:
```
from sklearn.metrics import confusion_matrix

my_cm=confusion_matrix(y_train,y_pred,labels=["0","
my_cm
```

Out[15]:
```
array([[5858,    0,    5,    0,    0,    4,   25,
0,   31,    0],
       [   1, 6667,   35,   16,    4,    5,    1,
6,    3,    4],
       [  16,    2, 5909,    3,    4,   17,    1,
1,    4,    1],
       [   0,    8,   83, 5951,    4,   22,    0,
18,   27,   18],
       [  20,    2,   15,    2, 5696,   26,    0,
46,    1,   34],
       [  14,    6,   15,   32,   23, 5227,   49,
0,   43,   12],
       [  33,    0,    4,    0,    0,   19, 5857,
0,    5,    0],
       [   1,    3,   45,    6,    8,    0,    0, 6
187,    0,   15],
       [  11,    2,   21,   49,    0,   26,   66,
0, 5671,    5],
       [   4,    1,    0,   50,   36,   14,    0,
82,   16, 5746]])
```

Question:

Look up the confusion matrix on the scikit-learn website, which axis is the true assignment and which is the prediction?

What are the three most common mistakes the classifier is making?

Thinking about the numbers involved, does this make sense?

Answer:

- Rows are the true values and the columns were the predicted
- When the model predicted a 2 incorrectly when the value was a 3, 82 times.
- When the model predicted a 9 incorrectly when the value was a 7, 82 times.
- When the model predicted a 3 incorrectly when the value was a 9, 50 times.

- Yes, 2 and 3, 9 and 7, 3 and 9 are similar to one another.

Accuracy is the number of correct predictions divided by the total number of assignments.

It is sort of the most common way to think about how a classifier is working, but it may not really be what you want to know.

If all mistakes are equally bad, then accuracy is reasonable metric of success.

But what if not all errors were equally bad? What if some errors were more costly than others?

but to proceed with calculationg accuracy from the confusion matric

The correct assigments are all along the diagonals, the wrong answers are the off diagonals

Accuracy= (sum along the diagonals)/(sum of all cells)

the sum along the diagonal is called the trace

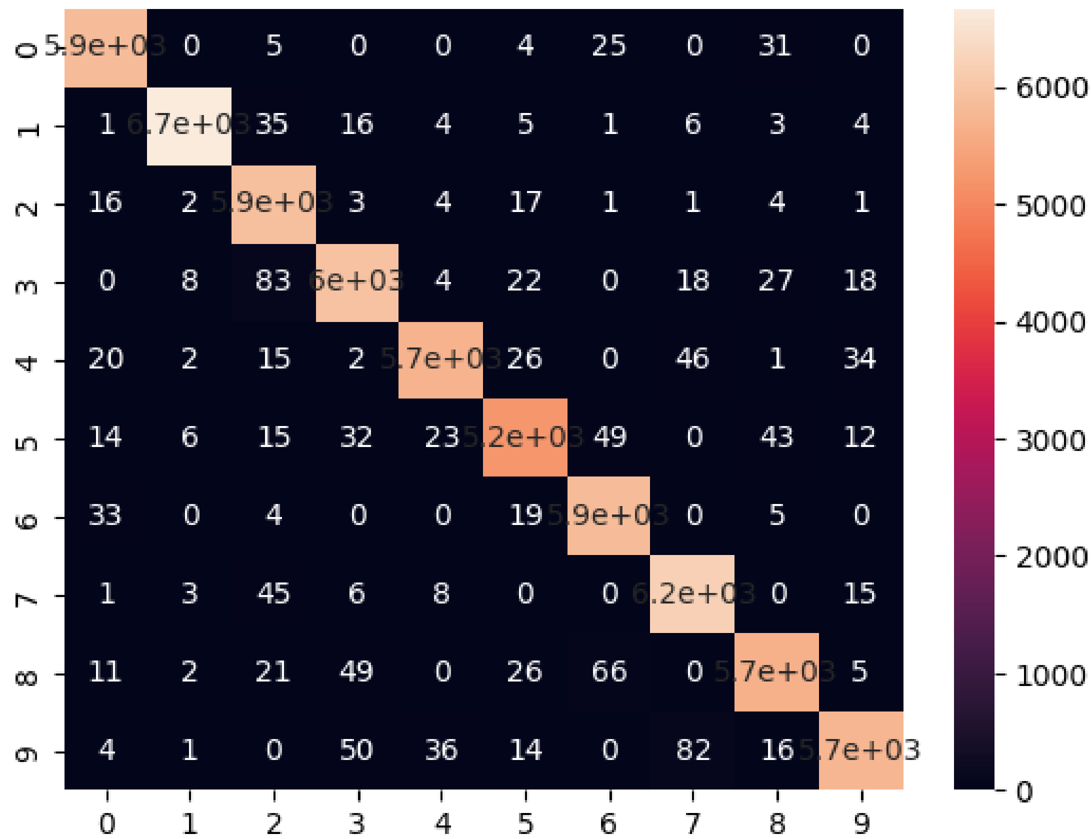
In [16]:
```
my_cm.trace()
```
Out[16]:
```
58769
```

In [17]:
```
my_cm.sum()
```
Out[17]:
```
60000
```

In [23]:
```
#Question-  write a formula that computes the accur

my_cm.trace()/my_cm.sum()
#97% accuracy
```
Out[23]:
```
0.9794833333333334
```

In [19]:
```
# Displaying the confusion matrix as a heat map

import seaborn as sns

sns.heatmap(my_cm,annot=True)
```

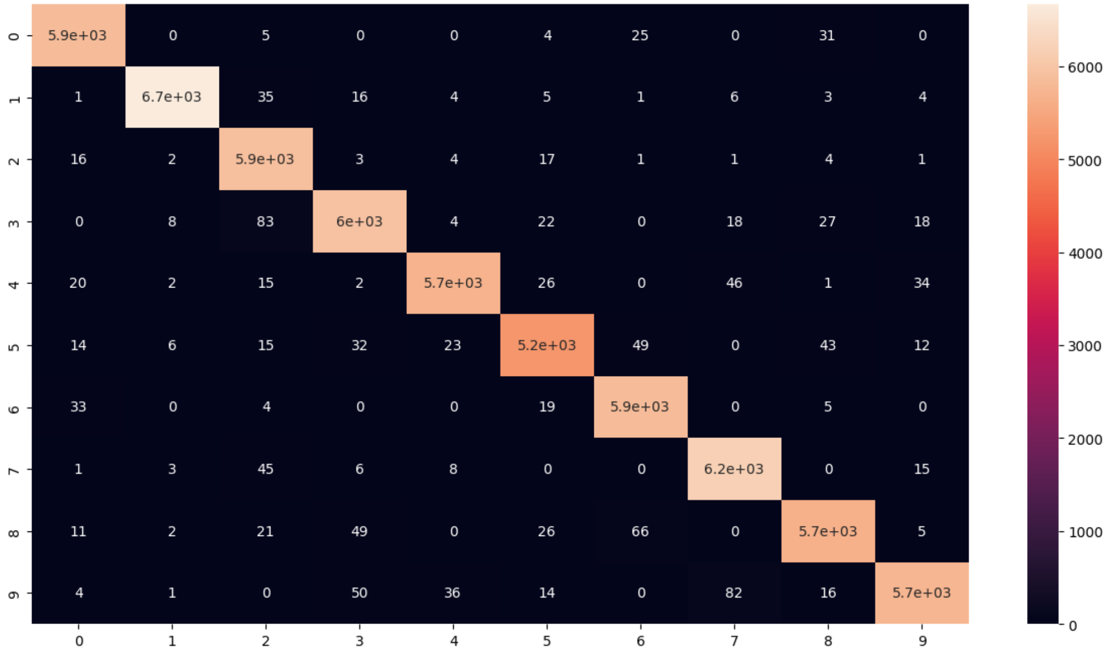
Out[19]: <Axes: >



In [20]:
```python
# controlling plot size-make the plot bigger so it
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [15, 8]
```

In [21]:
```python
sns.heatmap(my_cm,annot=True)
```

Out[21]: <Axes: >

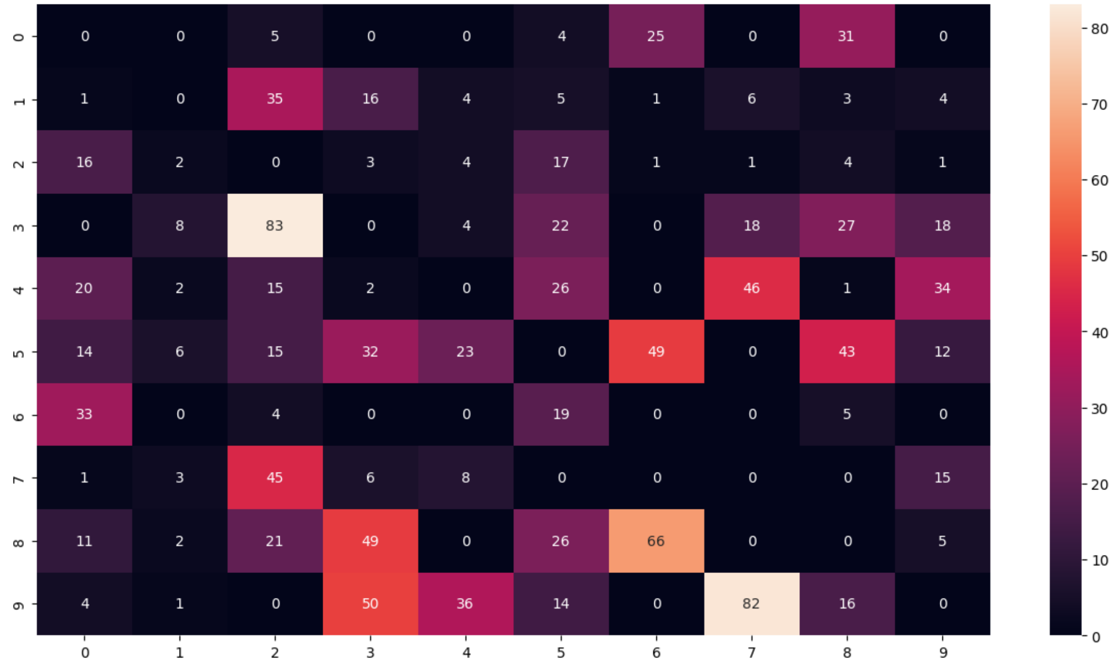


In [22]:

```
# The plot above is sort of disappointing,   the he
# it is hard to see how the off diagonals vary

# below,  we remove all the diagonal elements,  usi
#  my_cm-np.eye(my_cm.shape[0])*my_cm,

# np.eye(my_cm.shape[0])*my_cm cell by cell multipl
# diagonal elements of my_cm,  which we then subtra

import numpy as np

sns.heatmap(my_cm-np.eye(my_cm.shape[0])*my_cm,anno
```

Out[22]:

```
<Axes: >
```



What are the most prominent misclassifications?

Is this matrix symmeteric? Whould you expect it to be?

- When the model predicted a 2 incorrectly when the value was a 3, 82 times.
- When the model predicted a 9 incorrectly when the value was a 7, 82 times.
- When the model predicted a 3 incorrectly when the value was a 9, 50 times.

- the matrix is not symmetric because the Rows are the true values and the columns were the predicted