Beyond accuracy for full MNIST

We now have tools to *measure* weaknesses in our predictions.

But to become a successful Data Scientist, you will need to take some *action* to correct the mis-predictions.

We roll up our sleeves and diagnose errors

- Perform a deep dive into the misclassified examples
- Determine if there is a systematic problem
- Propose model improvements
    - Feature Engineering: adding features to help the model identify classes more easily
    - Modify the Loss function

We will focus on diagnosis for now and leave potential improvements for you to experiment with.

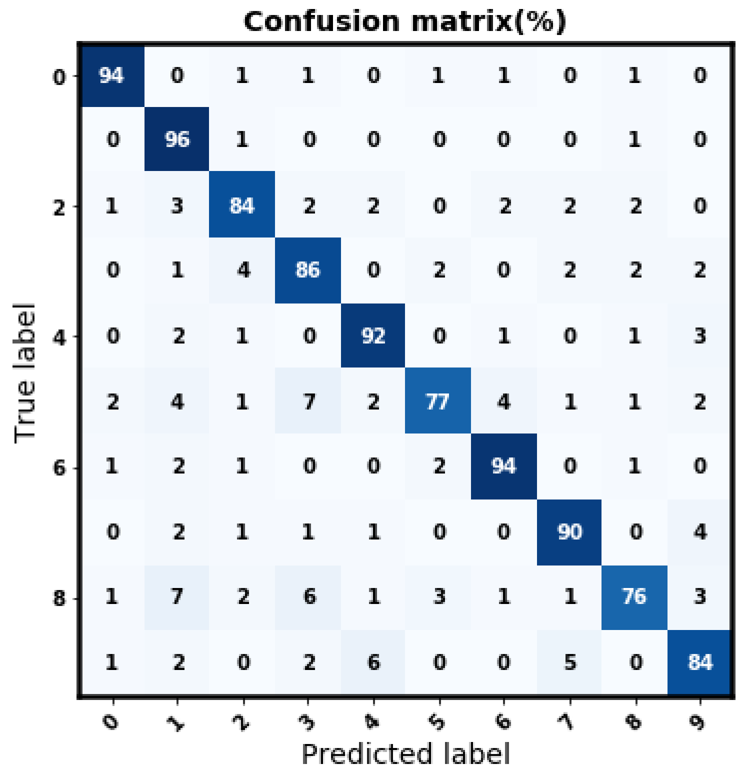
Let's start by fitting the model and plotting the Confusion Matrix.

In [5]:
```
mnh = mnist_helper.MNIST_Helper(random_seed=42)
mnh.setup()
_= mnh.fit()

# Now predict the value of the digit on the second half:
fig, ax = plt.subplots(figsize=(12,6))
confusion_mat = mnh.create_confusion_matrix()

digits = range(0,10)
_= clh.plot_confusion_matrix(confusion_mat, digits, ax=ax, normalize=True)
```

```
Retrieving MNIST_784 from cache
Normalized confusion matrix
```



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	94	0	1	1	0	1	1	0	1	0
1	0	96	1	0	0	0	0	0	1	0
2	1	3	84	2	2	0	2	2	2	0
3	0	1	4	86	0	2	0	2	2	2
4	0	2	1	0	92	0	1	0	1	3
5	2	4	1	7	2	77	4	1	1	2
6	1	2	1	0	0	2	94	0	1	0
7	0	2	1	1	1	0	0	90	0	4
8	1	7	2	6	1	3	1	1	76	3
9	1	2	0	2	6	0	0	5	0	84

What we have done above is to examine *Conditional Accuracy* (Recall) via the Confusion Matrix:

- Accuracy conditioned on examples with a particular target

Let's "zoom in" and examine mis-classified examples for a single digit $D \in \{0, 1, \ldots, 9\}$

- If we see some commonality on these examples, we might propose a synthetic feature to compensate

We can achieve this with a little coding

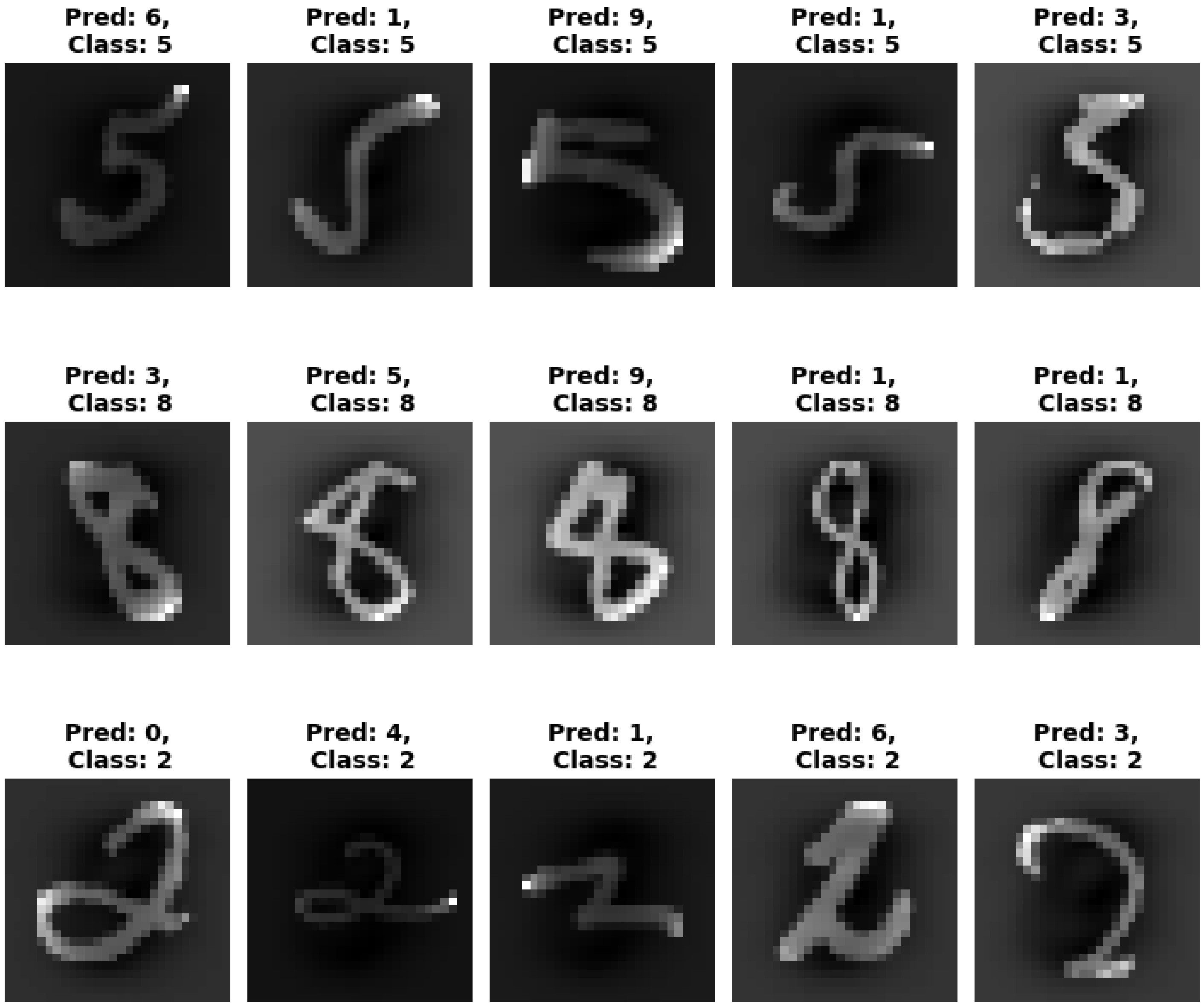
- Remember: Jupyter allows use to inspect the code by typing `mnh.plot_problem_digits??` into a code cell

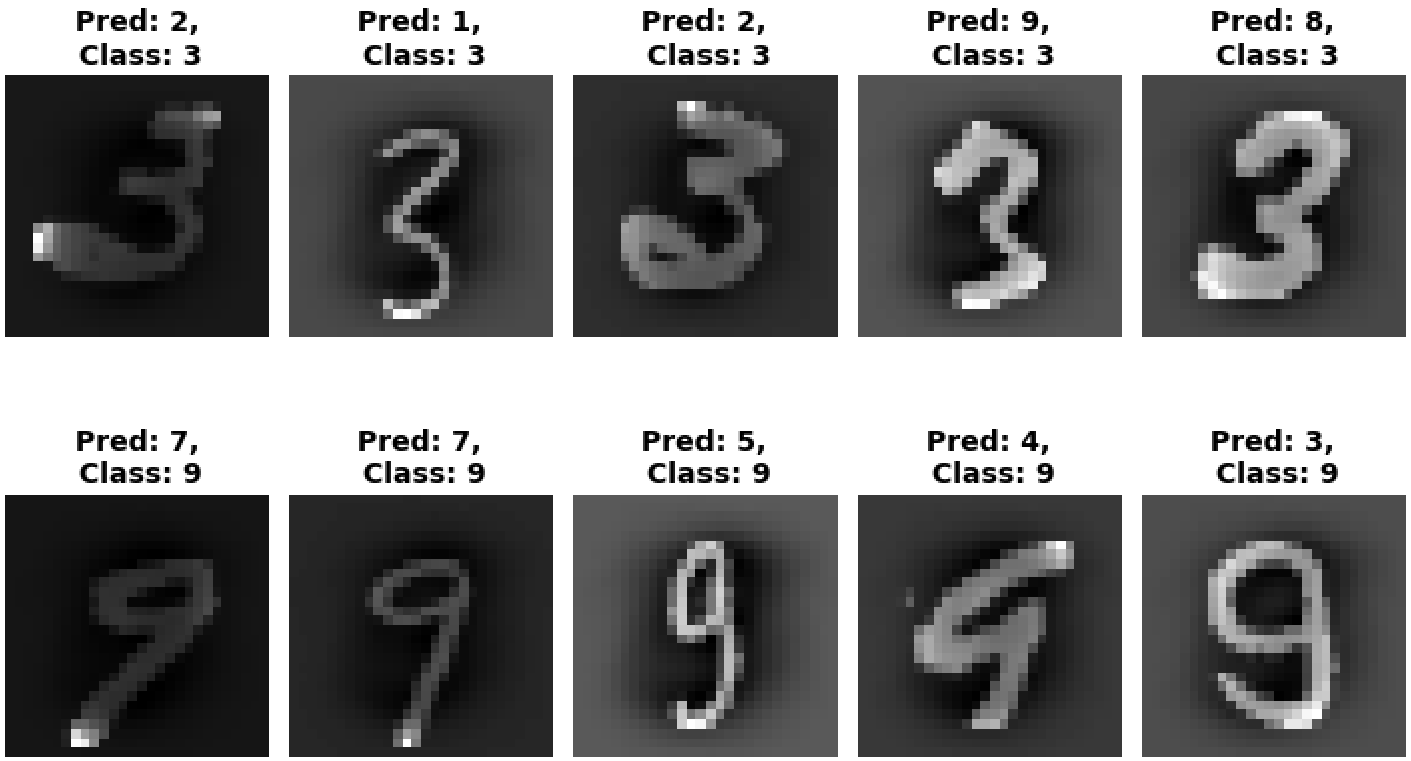
In [6]:

```
# Inspect the code by removing the leading "#" on the next line
# mnh.plot_problem_digits??
```

In [7]:
```python
problem_digits = [ '5', '8', '2', '3', '9' ]

mnh.plot_problem_digits( problem_digits )
```




Pred: 2, Class: 3
Pred: 1, Class: 3
Pred: 2, Class: 3
Pred: 9, Class: 3
Pred: 8, Class: 3
Pred: 7, Class: 9
Pred: 7, Class: 9
Pred: 5, Class: 9
Pred: 4, Class: 9
Pred: 3, Class: 9

The first row shows the misclassified 5's.

Why were they mis-classified ? Theory to test

- Are they of different proportion than correctly classified 5's ? (larger "tops" than "bottoms")

What about the misclassified 8's ? Theory to test

- Are the rightward "tilt" more extreme than correctly classified 8's ?

There is another possibility:

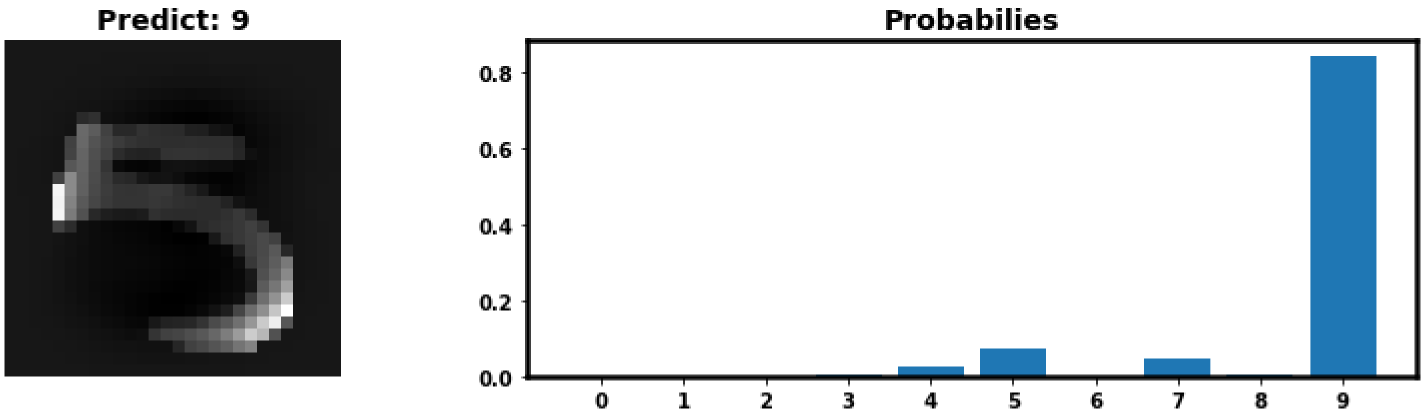
- With each example i there is a vector $\hat{p}^{(\mathbf{i})}$ (of length 10) of probabilities of the image being each digit
- We choose as the single prediction the digit with highest probability
$$\hat{\mathbf{y}}^{(\mathbf{i})} = \underset{c}{\mathbf{argmax}}\, \hat{p}_c^{(\mathbf{i})}$$
- Is it possible that the probability of the incorrect class was barely above that of the correct class ?

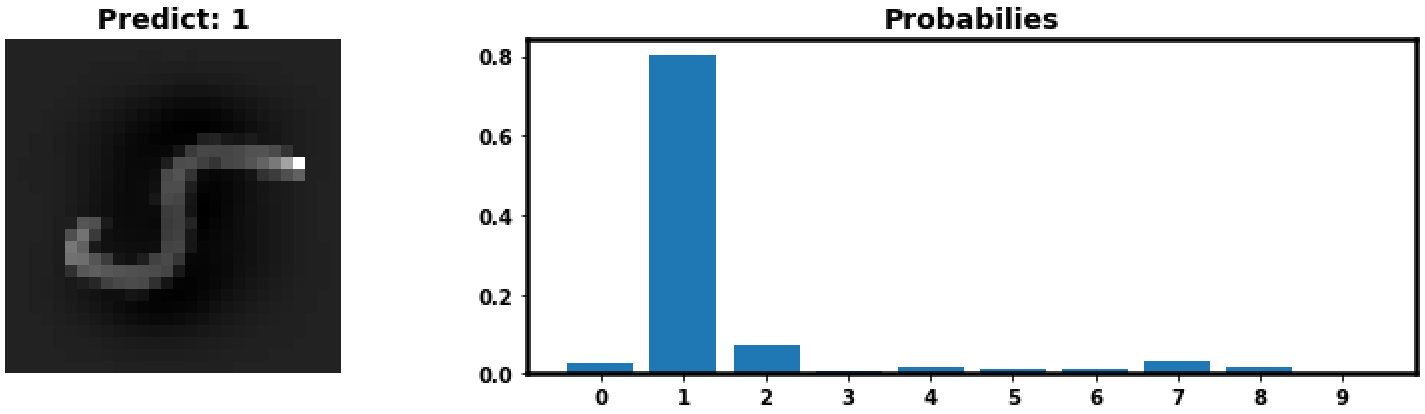
Let's answer this question with some code.

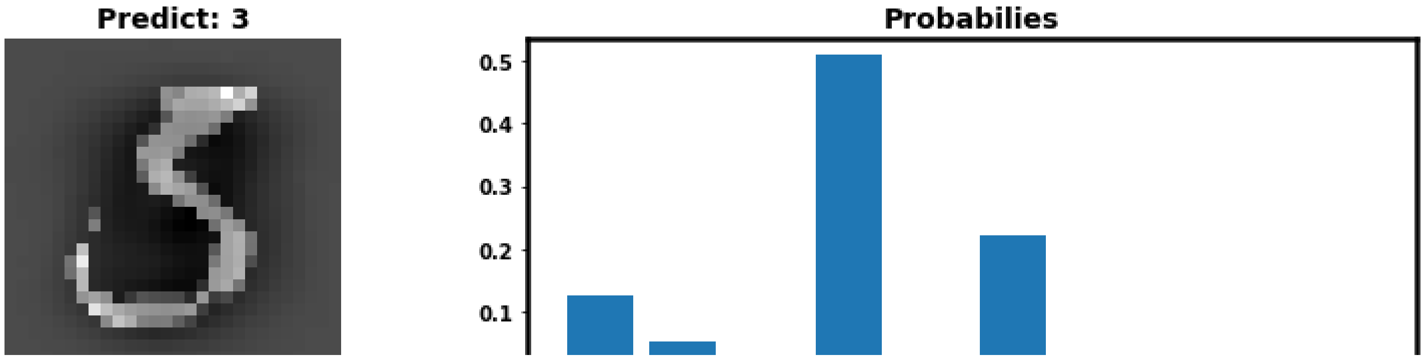
- Remember: Jupyter allows use to inspect the code by typing `mnh.predict_with_probs??` into a code cell

```
In [8]: # Inspect the code by removing the leading "#" on the next line
        # mnh.predict_with_probs??
```

In [9]:
```python
# Select some mis-classified "5"'s'
problems = mnh.misclassified["5"][ [2,3,4] ]
mnh.predict_with_probs(problems, digits)
```







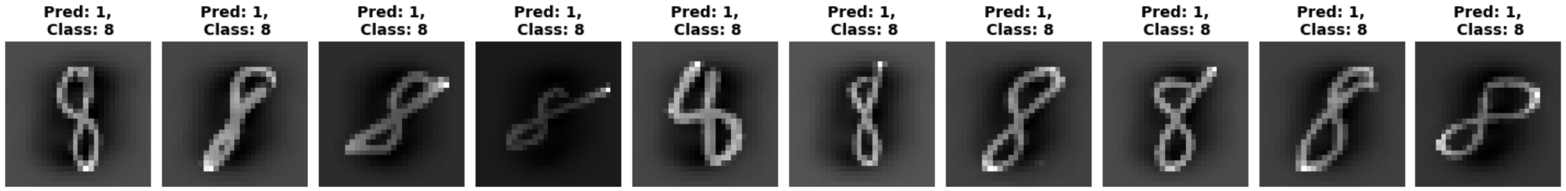
0.0
0
1
2
3
4
5
6
7
8
9

Examining the probabilities, we refute the possibility

- The probability of the correct class (5) is *much* lower than several incorrect classes

Let's narrow our focus to a single digit (8) with a misclassification to a particular digit (1)

```
In [10]: mnh.plot_problem_digits( ['8'], wrong_class='1', num_cols=10)
```



Theories:

- Are a couple of extremely bright pixels a possible cause for misclassification ?
- Is the "broken" or "mis-shaped" top a possible cause for misclassification ?

We leave it to you to explore these and alternative theories, along with possible fixes.

```
In [11]: print("Done")
```

```
Done
```