Demo: Logistic regression for classification of handwritten digits

In this demo, we will explore the use of logistic regression for classification of handwritten digits. In other words, given an image of a handwritten digit, we want to classify it as a 0, 1, 2, 3, …

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Load the digits dataset

For this demo, we will use a dataset known as MNIST. It contains 70,000 samples of handwritten digits, size-normalized and centered in a fixed-size image. Each sample is represented as a 28x28 pixel array, so there are 784 features per samples.

We will start by loading the dataset using the `fetch_openml` function. This function allows us to retrieve a dataset by name from OpenML, a public repository for machine learning data and experiments.

```
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
```

We observe that the data has 784 features and we have 70,000 samples:

```
X.shape
```

```
(70000, 784)
```

The target variables is a label for each digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. There are 6000-8000 samples for each class.

```
y.shape
```

```
(70000,)
```

```
print(y)
```

```
['5' '0' '4' ... '4' '5' '6']
```

```
pd.Series(y).value_counts()
```

```
1    7877
7    7293
3    7141
2    6990
9    6958
```

```
0    6903
6    6876
8    6825
4    6824
5    6313
dtype: int64
```

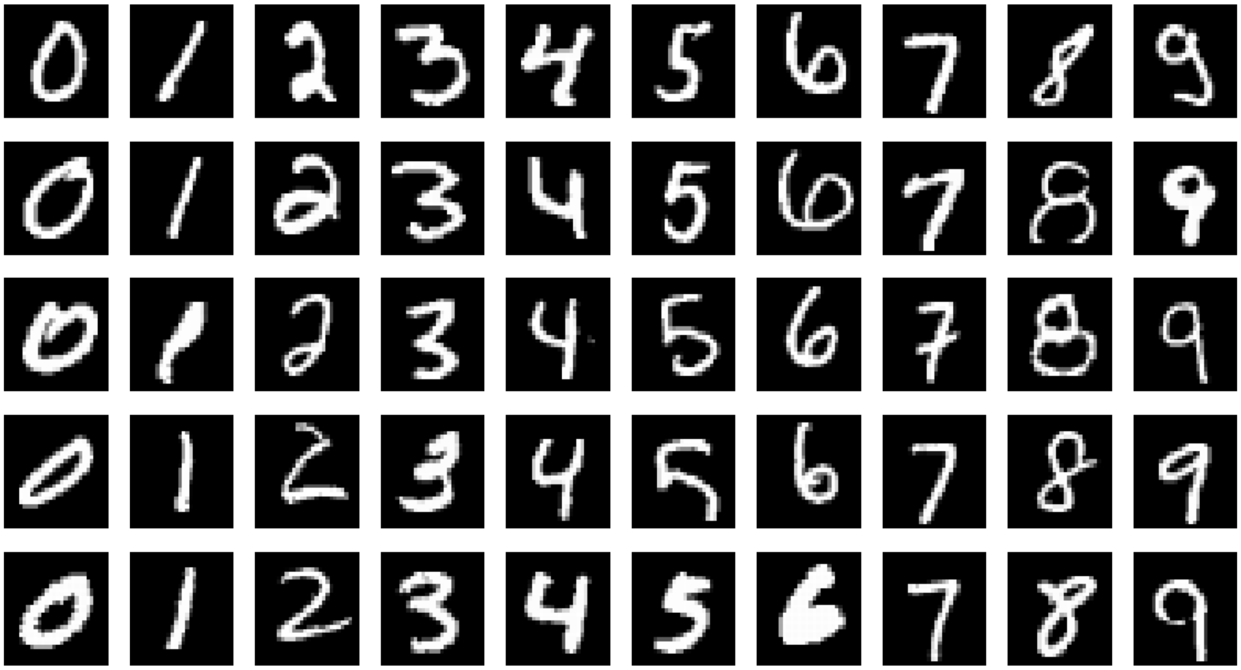
```
classes = ['0', '1', '2','3', '4','5', '6', '7', '8', '9']
nclasses = len(classes)
```

Each "feature" represents a pixel in the image, and each pixel can take on any integer value from 0 to 255. A large value for a pixel means that there is writing in that part of the image.

We can see a few examples, by plotting the 784 features as a 28x28 grid. In these images, white pixels indicate high values in the feature matrix.

```
samples_per_class = 5
figure = plt.figure(figsize=(nclasses*2,(1+samples_per_class*2)));

for idx_cls, cls in enumerate(classes):
  idxs = np.flatnonzero(y == cls)
  idxs = np.random.choice(idxs, samples_per_class, replace=False)
  for i, idx in enumerate(idxs):
    plt_idx = i * nclasses + idx_cls + 1
    p = plt.subplot(samples_per_class, nclasses, plt_idx);
    p = sns.heatmap(np.reshape(X[idx], (28,28)), cmap=plt.cm.gray,
             xticklabels=False, yticklabels=False, cbar=False);
    p = plt.axis('off');
```



Prepare data

Next, we will split our data into a test and training set.

We can use train_test_split from sklearn.model_selection to split the data.

Since the dataset is very large, it can take a long time to train a classifier on it. We just want to use it to demonstrate some useful concepts, so we will work with a smaller subset of the dataset. When we split the data using the train_test_split function, we will specify that we want 7,500 samples in the training set and 2,500 samples in the test set.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=9,
                                     train_size=7500, test_size=2500)
```

We'l also scale the data so that each feature takes on a value between 0 and 1.

```
X_train_scaled = X_train/255.0
X_test_scaled = X_test/255.0
```

Train a classifier using logistic regression

Finally, we are ready to train a classifier. We will use sklearn's LogisticRegression.

Unlike the linear regression, there is no closed form solution to the least squares parameter estimate in logistic regression. Therefore, we need to use a "solver" which finds a numerical solution. Several solvers are available for use with sklearn's LogisticRegression, but they don't all support all varieties of logistic regression.

We will use the saga solver, which

- works well when there is a large number of samples,
- supports logistic regression with no regularization penalty, L1 penalty, L2 penalty, or ElasticNet (which uses both penalties),
- and also supports multinomial regression with multiple classes, using the softmax function.

In addition to specifying which solver we want to use, we also specify a tolerance, which gives stopping criteria for the solver. A higher tolerance will finish faster, but may not find the optimal solution.

```
clf = LogisticRegression(penalty='none',
                         tol=0.1, solver='saga',
                         multi_class='multinomial').fit(X_train_scaled, y_train)
```

Once the classifier has been trained (fitted), we can get the coefficient values.

We had 784 features - one for each pixel - so we will have 784 coefficients. Furthermore, we have 10 classes, so we will have a vector of 784 coefficients for each of the 10 classes.

Therefore, our coefficient matrix has 10 rows and 784 columns:

```
clf.coef_.shape
```

```
(10, 784)
```

Interpret the coefficients of the logistic regression

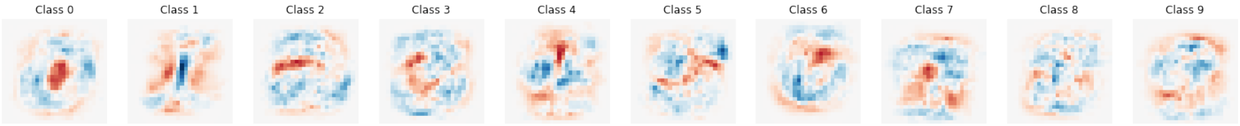
One benefit of logistic regression is its interpretability - we can use the coefficient values to understand what features (i.e. which pixels) are important in determining what class a sample belongs to.

The following plot shows the coefficient vector for each class, with positive coefficients in blue and negative coefficients in red.

```
scale = np.max(np.abs(clf.coef_))

p = plt.figure(figsize=(25, 2.5));

for i in range(nclasses):
    p = plt.subplot(1, nclasses, i + 1)
    p = plt.imshow(clf.coef_[i].reshape(28, 28),
                  cmap=plt.cm.RdBu, vmin=-scale, vmax=scale);
    p = plt.axis('off')
    p = plt.title('Class %i' % i);
```



We can see which pixels are positively associated with belonging to the class, and which pixels are negatively associated with belonging to the class.

For example, consider Class 0. If a sample has large values in the pixels shown in blue (the 0 shape around the center of the image), the probability of that sample being a 0 digit increases. If the sample has large values in the pixels in the center of the image, the probability of the sample being a 0 digit decreases.

Many pixels have coefficients whose magnitude are very small. These are shown in white, and they are not very important for this classification task.

Use a fitted logistic regression

Given the coefficient matrix, we can get the per-class probability for any sample.

We know that for logistic regression with the softmax function, the conditional probability of a sample belonging to class k is given by:

$$P(y=k|\text{x}) = \frac{e^{z_k}}{\sum_{\ell=1}^{K} e^{z_\ell}}$$

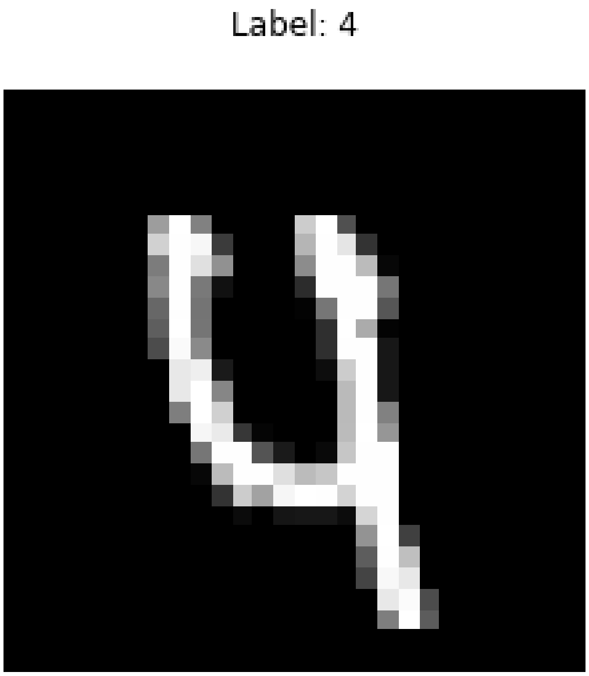
where $z_k = w_k x$.

(w_k is the weight vector for class k, and x includes a 1s column so that the intercept can be included in the weight matrix.)

As an example, let's look at a specific test sample:

```
sample_idx = 33
```

```
plt.imshow(X_test_scaled[sample_idx].reshape(28,28), cmap='gray');
plt.title('Label: %s\n' % y_test[sample_idx]);
plt.axis('off');
```



We'l compute z_k for each class k:

```
z = [ clf.intercept_[k] + np.dot(clf.coef_[k], X_test_scaled[sample_idx]) for k in range(10)
    ]
z
```

```
[-2.637557861149052,
 -4.170443056448888,
 2.442377593434767,
 0.0064046677157321985,
 2.2737701179833265,
 -2.2273963841248268,
 -0.570398496427391,
 2.2229340608118875,
 -1.282401148186226,
 3.9427105063906764]
```

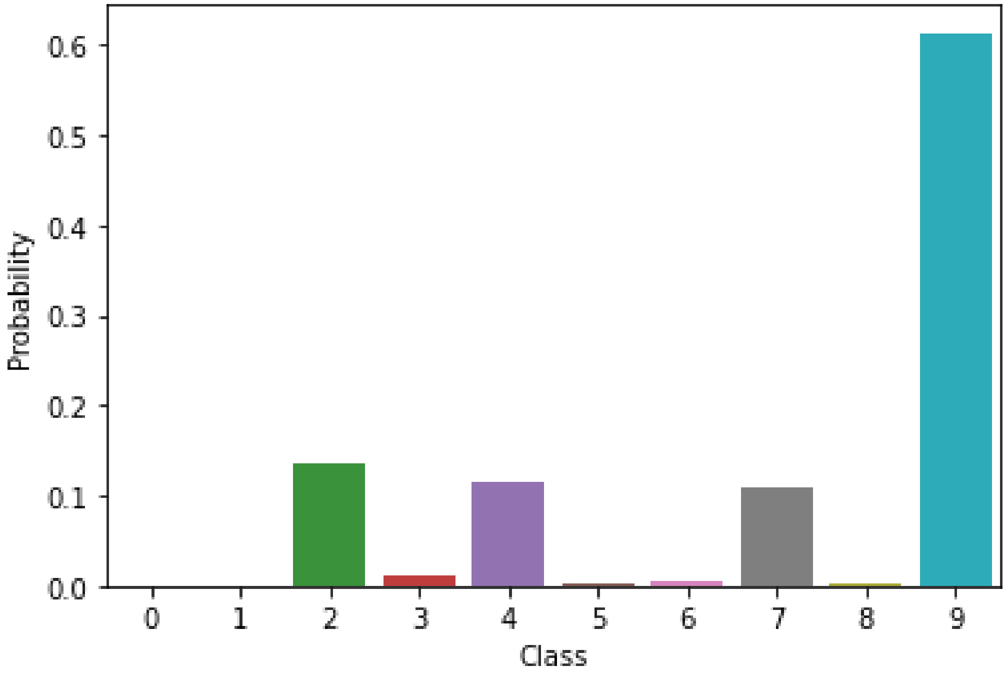
Then, we can compute the conditional probability for each class, for this sample:

```
exps = [np.exp(z[k]) for k in range(10)]
exps_sum = np.sum(exps)
probs = exps/exps_sum
probs
```

```
array([8.51081737e-04, 1.83758607e-04, 1.36823031e-01, 1.19737340e-02,
       1.15593632e-01, 1.28263220e-03, 6.72554025e-03, 1.09864173e-01,
       3.29995755e-03, 6.13402459e-01])
```

Here, the first entry is the probability of belonging to class 0 (i.e. having the label '0'), the second entry is the probability of belonging to class 1, etc.

```
sns.barplot(x=np.arange(0,10), y=probs);
plt.ylabel("Probability");
plt.xlabel("Class");
```



In general, to get the predicted *label*, we can find the class with the highest probability:

```
idx_cls = np.argmax(probs)
classes[idx_cls]
```

```
'9'
```

If this matches the actual label for the first test sample, then our prediction is correct.

```
y_test[sample_idx]
```

```
'4'
```

The `LogisticRegression` implementation in `sklearn` includes functions to compute both the per-class probability, and the most likely label.

We can use the `predict_proba` function on the logistic regression to get these probabilities. For each sample, it returns 10 probabilities - one for each of the ten classes (i.e. each value of k).

```
y_pred_prob = clf.predict_proba(X_test_scaled)
```

Let's look at our example test point, and compare to our own computations:

```
y_pred_prob[sample_idx]
```

```
array([8.51081737e-04, 1.83758607e-04, 1.36823031e-01, 1.19737340e-02,
       1.15593632e-01, 1.28263220e-03, 6.72554025e-03, 1.09864173e-01,
       3.29995755e-03, 6.13402459e-01])
```

We use the `predict` function to predict a label for each sample in the test set. This will return the class label with the highest probability.

For our test sample, the prediction is:

```
y_pred = clf.predict(X_test_scaled)
```

```
y_pred[sample_idx]
```

```
'9'
```

and the true value is:

```
y_test[sample_idx]
```

```
'4'
```

Evaluate classifier performance

The first important metric is the accuracy - what percent of predicted labels are the same as the true labels?

There are a few ways to compute this value -

```
accuracy =  np.mean(y_test == y_pred)
print(accuracy)
```

```
0.9044
```

```
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
```

```
0.9044
```

```
accuracy = clf.score(X_test_scaled, y_test)
print(accuracy)
```

```
0.9044
```

What about other important metrics?

For a binary classifier, we also care about

- The number of true positive (TP) outputs - samples from the positive class that are predicted as positive
- The number of true negative (TN) outputs - samples from the negative class that are predicted as negative

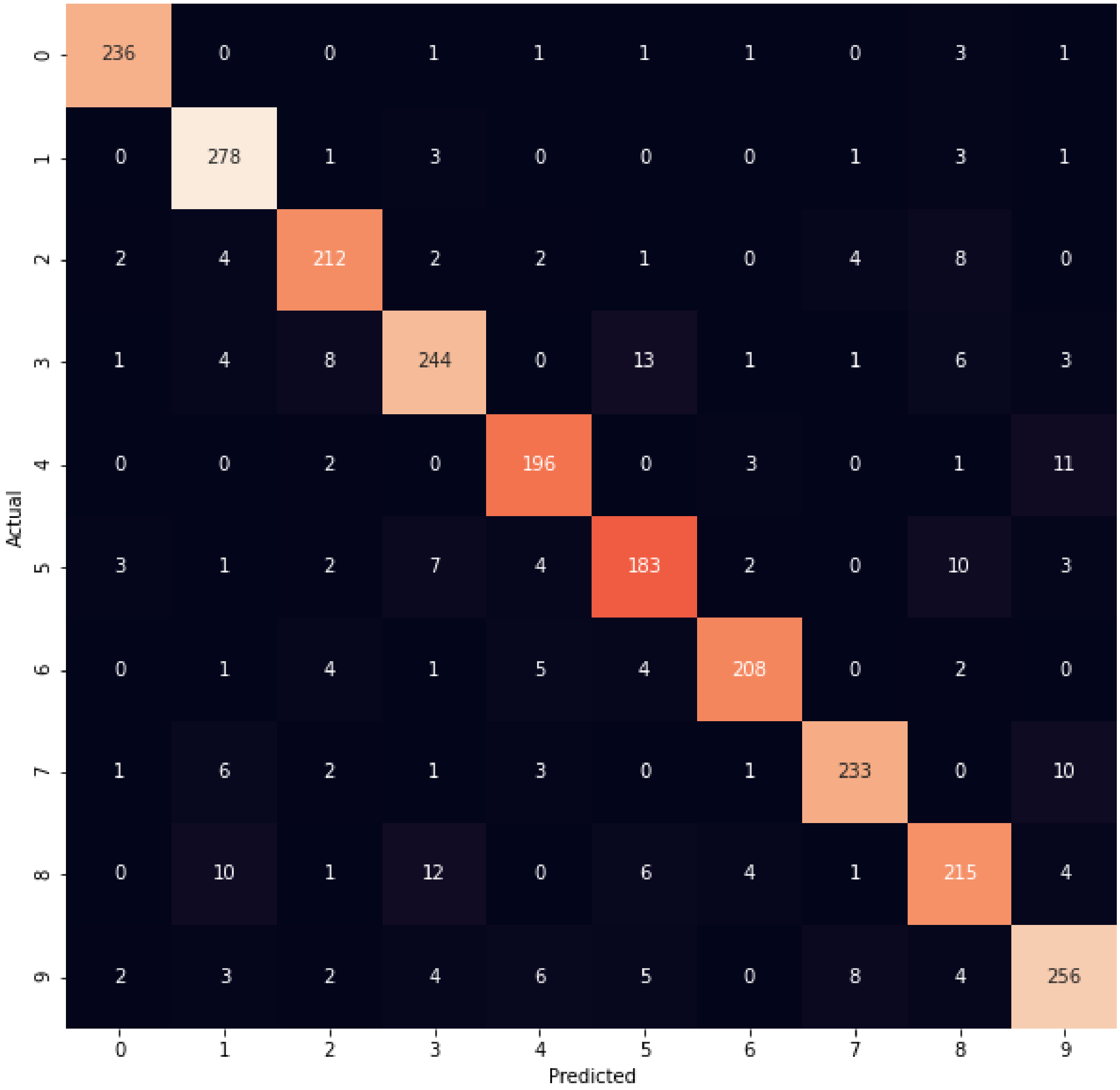
- The number of false positive (FP) outputs - samples from the negative class that are predicted as positive, and
- The number of false negative (FN) outputs - samples from the positive class that are predicted as negative.

These are often presented together in a confusion matrix.

For a multi-class problem, we can extend the confusion matrix to have more rows and columns. The diagonal of the multi-class confusion matrix shows the number of correct classifications for each class, and other entries show instances where a sample from one class was mistakenly assigned a different class label.

We can create a confusion matrix using the `pandas` library's `crosstab` function.

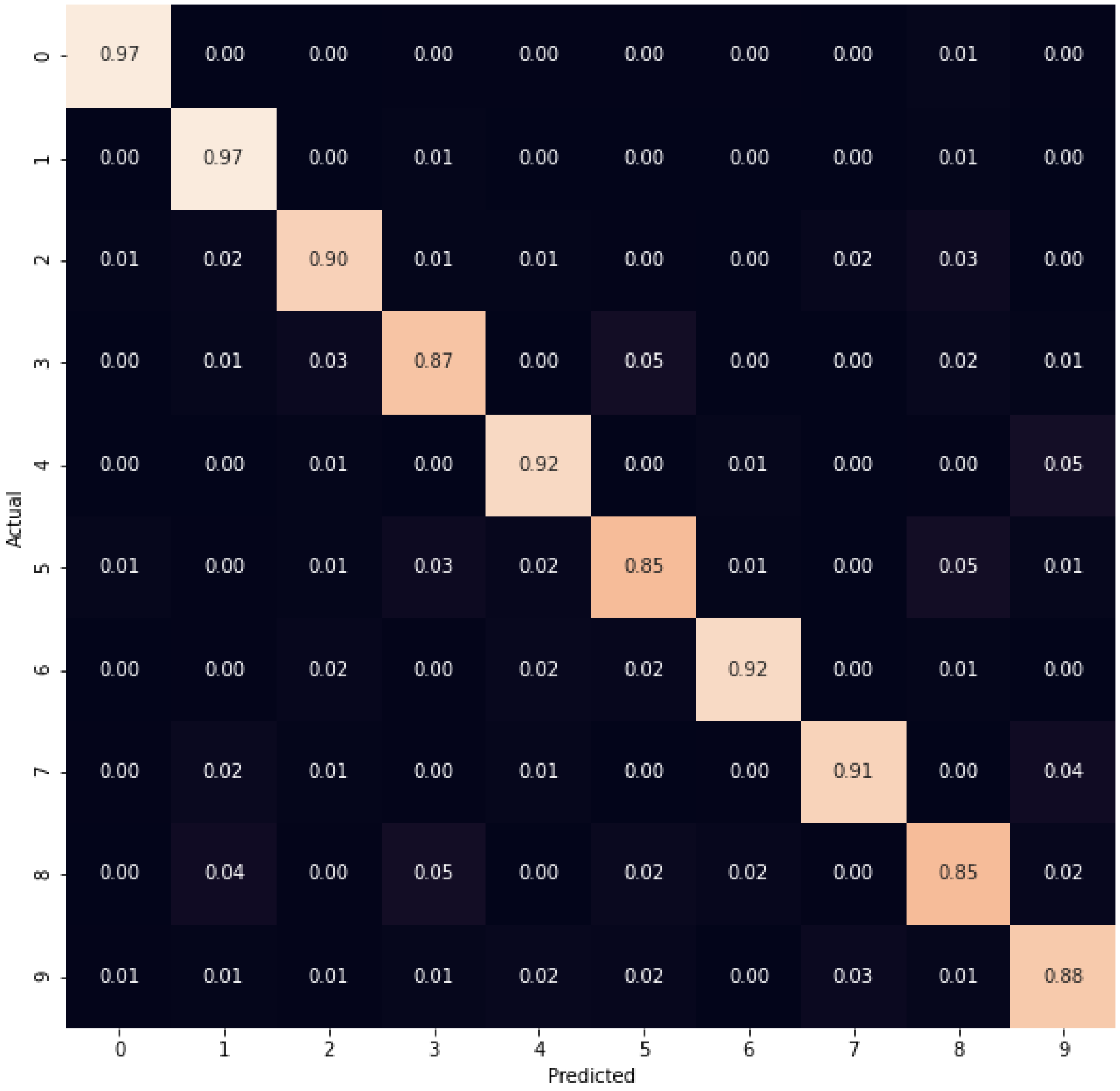
```
cm = pd.crosstab(y_test, y_pred,
                               rownames=['Actual'], colnames=['Predicted'])
p = plt.figure(figsize=(10,10));
p = sns.heatmap(cm, annot=True, fmt="d", cbar=False)
```



Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	236	0	0	1	1	1	1	0	3	1
1	0	278	1	3	0	0	0	1	3	1
2	2	4	212	2	2	1	0	4	8	0
3	1	4	8	244	0	13	1	1	6	3
4	0	0	2	0	196	0	3	0	1	11
5	3	1	2	7	4	183	2	0	10	3
6	0	1	4	1	5	4	208	0	2	0
7	1	6	2	1	3	0	1	233	0	10
8	0	10	1	12	0	6	4	1	215	4
9	2	3	2	4	6	5	0	8	4	256

Here’s a version that is slightly easier to interpret - we have normalized the confusion matrix by row, so that the entries on the diagonal show the accuracy per class.

```
cm = pd.crosstab(y_test, y_pred,
                               rownames=['Actual'], colnames=['Predicted'],
                                   normalize='index')
p = plt.figure(figsize=(10,10));
p = sns.heatmap(cm, annot=True, fmt=".2f", cbar=False)
```



	0	1	2	3	4	5	6	7	8	9
0	0.97	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
1	0.00	0.97	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.00
2	0.01	0.02	0.90	0.01	0.01	0.00	0.00	0.02	0.03	0.00
3	0.00	0.01	0.03	0.87	0.00	0.05	0.00	0.00	0.02	0.01
4	0.00	0.00	0.01	0.00	0.92	0.00	0.01	0.00	0.00	0.05
5	0.01	0.00	0.01	0.03	0.02	0.85	0.01	0.00	0.05	0.01
6	0.00	0.00	0.02	0.00	0.02	0.02	0.92	0.00	0.01	0.00
7	0.00	0.02	0.01	0.00	0.01	0.00	0.00	0.91	0.00	0.04
8	0.00	0.04	0.00	0.05	0.00	0.02	0.02	0.00	0.85	0.02
9	0.01	0.01	0.01	0.01	0.02	0.02	0.00	0.03	0.01	0.88

We can see that the digits 0, 1, 4 are easiest for the logistic regression to classify, while the digits 8, 5, 2, and 3 are more difficult (because the classification accuracay was less for these digits).

We can also see which digits are easily confused with one another. For example, we can see that some 8s are misclassified as 1s, and some 5s are misclassified as 8s.