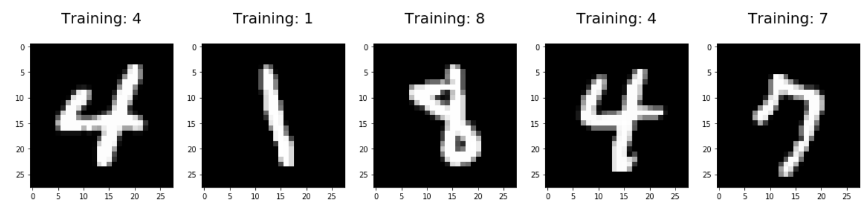
Logistic Regression using Python (scikit-learn)

Michael Galarnyk Follow
Sep 13, 2017 · 8 min read



Visualizing the Images and Labels in the MNIST Dataset

One of the most amazing things about Python's scikit-learn library is that is has a 4-step modeling pattern that makes it easy to code a machine learning classifier. While this tutorial uses a classifier called Logistic Regression, the coding process in this tutorial applies to other classifiers in sklearn (Decision Tree, K-Nearest Neighbors etc). In this tutorial, we use Logistic Regression to predict digit labels based on images. The image above shows a bunch of training digits (observations) from the MNIST dataset whose category membership is known (labels 0–9). After training a model with logistic regression, it can be used to predict an image label (labels 0–9) given an image.

Logistic Regression using Python Video

The first part of this tutorial post goes over a toy dataset (digits dataset) to show quickly illustrate scikit-learn's 4 step modeling pattern and show the behavior of the logistic regression algorthm. The second part of the tutorial goes over a more realistic dataset (MNIST dataset) to briefly show how changing a model's default parameters can effect performance (both in timing and accuracy of the model).
With that, lets get started. If you get lost, I recommend opening the video above in a separate tab. The code used in this tutorial is available below

Digits Logistic Regression (first part of tutorial code)

MNIST Logistic Regression (second part of tutorial code)

Getting Started (Prerequisites)

If you already have anaconda installed, skip to the next section. I recommend having anaconda installed (either Python 2 or 3 works well for this tutorial) so you won't have any issue importing libraries.

You can either download anaconda from the official site and install on your own or you can follow these anaconda installation tutorials below to set up anaconda on your operating system.

Install Anaconda on Windows: Link

Install Anaconda on Mac: Link

Install Anaconda on Ubuntu (Linux): Link

Logistic Regression on Digits Dataset

Loading the Data (Digits Dataset)

The digits dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below will load the digits dataset.

```
from sklearn.datasets import load_digits
digits = load_digits()
```

Now that you have the dataset loaded you can use the commands below

```
# Print to show there are 1797 images (8 by 8 images for a
dimensionality of 64)
print(“Image Data Shape” , digits.data.shape)

# Print to show there are 1797 labels (integers from 0–9)
print("Label Data Shape", digits.target.shape)
```

to see that there are 1797 images and 1797 labels in the dataset

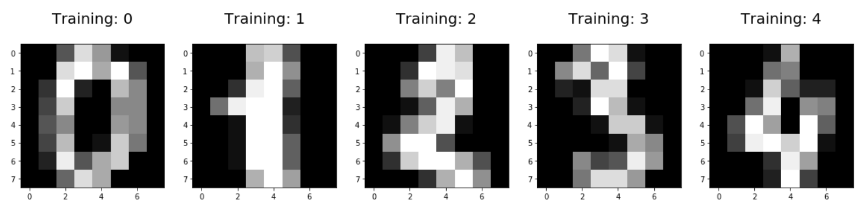
Showing the Images and the Labels (Digits Dataset)

This section is really just to show what the images and labels look like. It usually helps to visualize your data to see what you are working with.

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(digits.data[0:5],
digits.target[0:5])):
 plt.subplot(1, 5, index + 1)
```

```
    plt.imshow(np.reshape(image, (8,8)), cmap=plt.cm.gray)
    plt.title('Training: %i\n' % label, fontsize = 20)
```



Visualizing the Images and Labels in our Dataset

Splitting Data into Training and Test Sets (Digits Dataset)

We make training and test sets to make sure that after we train our classification algorithm, it is able to generalize well to new data.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test =
train_test_split(digits.data, digits.target, test_size=0.25,
random_state=0)
```

Scikit-learn 4-Step Modeling Pattern (Digits Dataset)

Step 1. Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

```
from sklearn.linear_model import LogisticRegression
```

Step 2. Make an instance of the Model

```
# all parameters not specified are set to their defaults
logisticRegr = LogisticRegression()
```

Step 3. Training the model on the data, storing the information learned from the data

Model is learning the relationship between digits (x_train) and labels (y_train)

```
logisticRegr.fit(x_train, y_train)
```

Step 4. Predict labels for new data (new images)

Uses the information the model learned during the model training process

```
# Returns a NumPy Array
# Predict for One Observation (image)
logisticRegr.predict(x_test[0].reshape(1,-1))
```

Predict for Multiple Observations (images) at Once

```
logisticRegr.predict(x_test[0:10])
```

Make predictions on entire test data

```
predictions = logisticRegr.predict(x_test)
```

Measuring Model Performance (Digits Dataset)

While there are other ways of measuring model performance (precision, recall, F1 Score, ROC Curve, etc), we are going to keep this simple and use accuracy as our metric.
To do this are going to see how the model performs on the new data (test set)

accuracy is defined as:

(fraction of correct predictions): correct predictions / total number of data points

```
# Use score method to get accuracy of model
score = logisticRegr.score(x_test, y_test)
print(score)
```

Our accuracy was 95.3%.

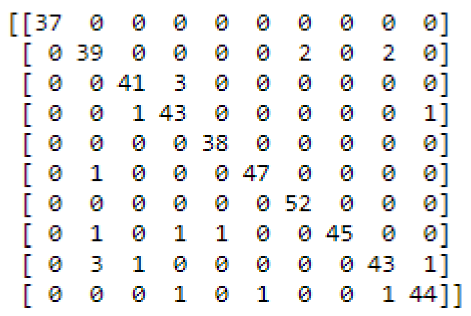
Confusion Matrix (Digits Dataset)

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. In this section, I am just showing two python packages (Seaborn and Matplotlib) for making confusion matrices more understandable and visually appealing.

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import metrics
```

The confusion matrix below is not visually super informative or visually appealing.

```
cm = metrics.confusion_matrix(y_test, predictions)
print(cm)
```



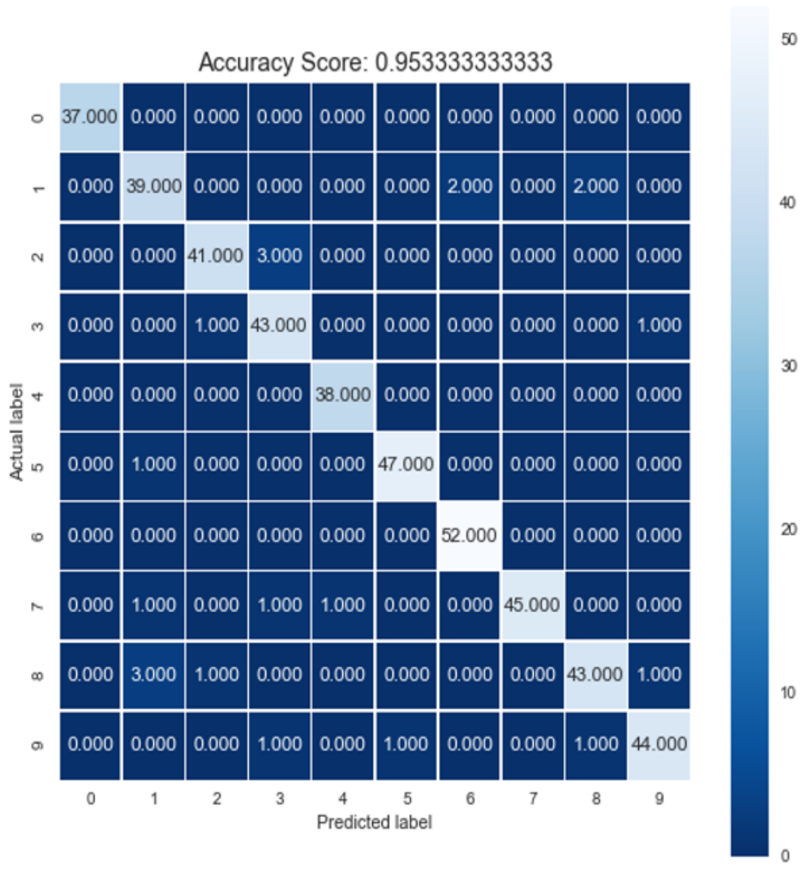
```
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

Not a visually appealing way to view a confusion matrix

Method 1 (Seaborn)

As you can see below, this method produces a more understandable and visually readable confusion matrix using seaborn.

```
plt.figure(figsize=(9,9))
sns.heatmap(cm, annot=True, fmt=".3f", linewidths=.5, square
= True, cmap = 'Blues_r');
plt.ylabel('Actual label');
plt.xlabel('Predicted label');
all_sample_title = 'Accuracy Score: {0}'.format(score)
plt.title(all_sample_title, size = 15);
```



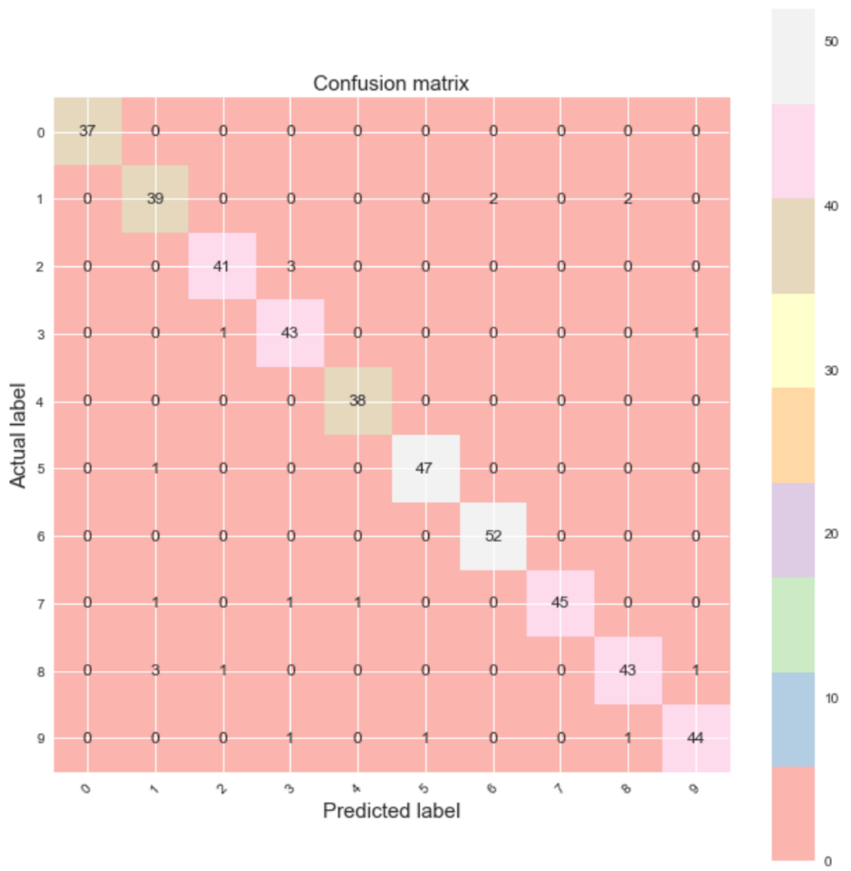
	0	1	2	3	4	5	6	7	8	9
0	37.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1	0.000	39.000	0.000	0.000	0.000	0.000	2.000	0.000	2.000	0.000
2	0.000	0.000	41.000	3.000	0.000	0.000	0.000	0.000	0.000	0.000
3	0.000	0.000	1.000	43.000	0.000	0.000	0.000	0.000	0.000	1.000
4	0.000	0.000	0.000	0.000	38.000	0.000	0.000	0.000	0.000	0.000
5	0.000	1.000	0.000	0.000	0.000	47.000	0.000	0.000	0.000	0.000
6	0.000	0.000	0.000	0.000	0.000	0.000	52.000	0.000	0.000	0.000
7	0.000	1.000	0.000	1.000	1.000	0.000	0.000	45.000	0.000	0.000
8	0.000	3.000	1.000	0.000	0.000	0.000	0.000	0.000	43.000	1.000
9	0.000	0.000	0.000	1.000	0.000	1.000	0.000	0.000	1.000	44.000

Confusion Matrix using Seaborn

Method 2 (Matplotlib)
This method is clearly a lot more code. I just wanted to show people how to do it in matplotlib as well.

```
plt.figure(figsize=(9,9))
plt.imshow(cm, interpolation='nearest', cmap='Pastel1')
plt.title('Confusion matrix', size = 15)
plt.colorbar()
tick_marks = np.arange(10)
plt.xticks(tick_marks, ["0", "1", "2", "3", "4", "5", "6",
"7", "8", "9"], rotation=45, size = 10)
plt.yticks(tick_marks, ["0", "1", "2", "3", "4", "5", "6",
"7", "8", "9"], size = 10)
plt.tight_layout()
plt.ylabel('Actual label', size = 15)
plt.xlabel('Predicted label', size = 15)
width, height = cm.shape


for x in xrange(width):
 for y in xrange(height):
  plt.annotate(str(cm[x][y]), xy=(y, x),
  horizontalalignment='center',
  verticalalignment='center')
```



Confusion Matrix using Matplotlib

Logistic Regression (MNIST)

One important point to emphasize that the digit dataset contained in sklearn is too small to be representative of a real world machine learning task.
We are going to use the MNIST dataset because it is for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. One of the things we will notice is that parameter tuning can greatly speed up a machine learning algorithm's training time.

Downloading the Data (MNIST)

The MNIST dataset doesn't come from within scikit-learn

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
```

Now that you have the dataset loaded you can use the commands below

```
# These are the images
# There are 70,000 images (28 by 28 images for a
dimensionality of 784)
print(mnist.data.shape)


# These are the labels
print(mnist.target.shape)
```

to see that there are 70000 images and 70000 labels in the dataset

Splitting Data into Training and Test Sets (MNIST)

The code below splits the data into training and test data sets. The `test_size=1/7.0` makes the training set size 60,000 images and the test set size of 10,000.

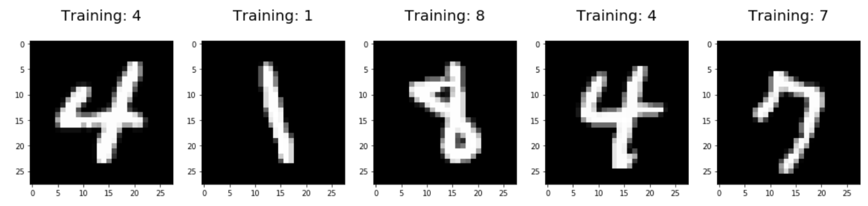
```
from sklearn.model_selection import train_test_split


train_img, test_img, train_lbl, test_lbl = train_test_split(
 mnist.data, mnist.target, test_size=1/7.0, random_state=0)
```

Showing the Images and Labels (MNIST)

```
import numpy as np
import matplotlib.pyplot as plt


plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(train_img[0:5],
train_lbl[0:5])):
 plt.subplot(1, 5, index + 1)
 plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
 plt.title('Training: %i\n' % label, fontsize = 20)
```



Visualizing the Images and Labels in our Dataset

Scikit-learn 4-Step Modeling Pattern (MNIST)

One thing I like to mention is the importance of parameter tuning. While it may not have mattered much for the smaller digits dataset, it makes a bigger difference on larger and more complex datasets. While usually one adjusts parameters for the sake of accuracy, in the case below, we are adjusting the parameter solver to speed up the fitting of the model.

Step 1. Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

```
from sklearn.linear_model import LogisticRegression
```

Step 2. Make an instance of the Model

Please see the documentation if you are curious what changing solver does. Essentially, we are changing the optimization algorithm.

```
# all parameters not specified are set to their defaults
# default solver is incredibly slow thats why we change it
logisticRegr = LogisticRegression(solver = 'lbfgs')
```

Step 3. Training the model on the data, storing the information learned from the data

Model is learning the relationship between x (digits) and y (labels)

```
logisticRegr.fit(train_img, train_lbl)
```

Step 4. Predict the labels of new data (new images)
Uses the information the model learned during the model training process

```
# Returns a NumPy Array
# Predict for One Observation (image)
logisticRegr.predict(test_img[0].reshape(1,-1))
```

Predict for Multiple Observations (images) at Once

```
logisticRegr.predict(test_img[0:10])
```

Make predictions on entire test data

```
predictions = logisticRegr.predict(test_img)
```

Measuring Model Performance (MNIST)

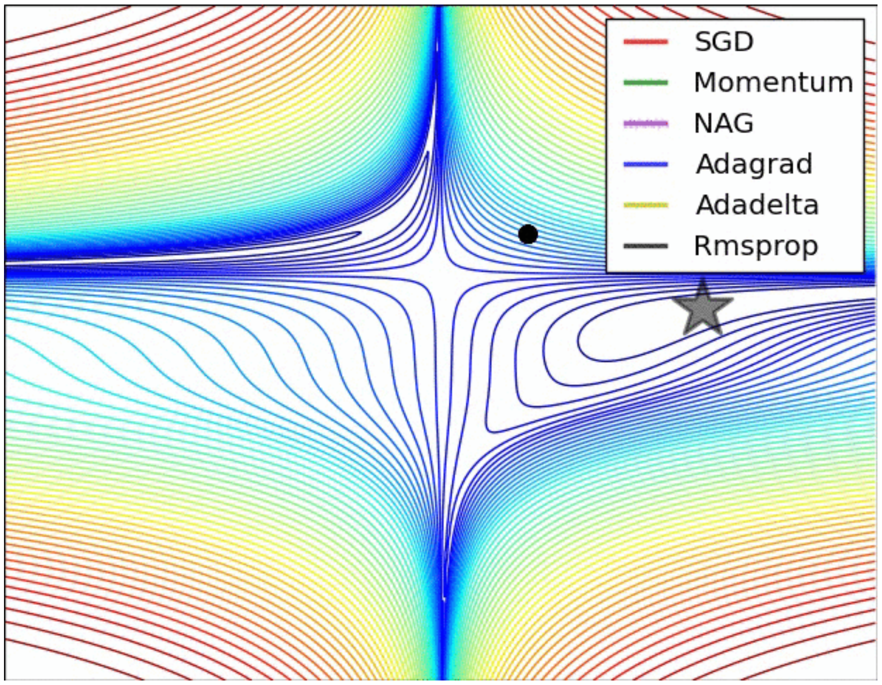
While there are other ways of measuring model performance (precision, recall, F1 Score, ROC Curve, etc), we are going to keep this simple and use accuracy as our metric.
To do this are going to see how the model performs on the new data (test set)

accuracy is defined as:

(fraction of correct predictions): correct predictions / total number of data points

```
score = logisticRegr.score(test_img, test_lbl)
print(score)
```

One thing I briefly want to mention is that is the default optimization algorithm parameter was `solver = liblinear` and it took 2893.1 seconds to run with a accuracy of 91.45%. When I set `solver = lbfgs` , it took 52.86 seconds to run with an accuracy of 91.3%. Changing the solver had a minor effect on accuracy, but at least it was a lot faster.



This gif just shows that some optimization algorithms take longer (image source)

Display Misclassified images with Predicted Labels (MNIST)

While I could show another confusion matrix, I figured people would rather see misclassified images on the off chance someone finds it interesting.

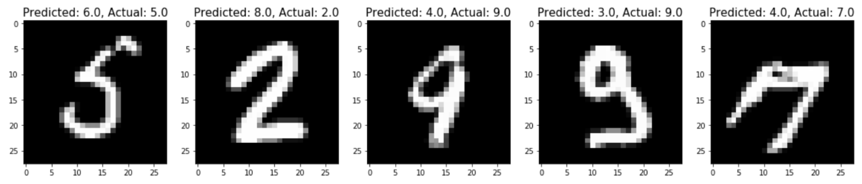
Getting the misclassified images' index

```
import numpy as np
import matplotlib.pyplot as plt
```

```
index = 0
misclassifiedIndexes = []
for label, predict in zip(test_lbl, predictions):
 if label != predict:
  misclassifiedIndexes.append(index)
  index +=1
```

Showing the misclassified images and image labels using matplotlib

```
plt.figure(figsize=(20,4))
for plotIndex, badIndex in
enumerate(misclassifiedIndexes[0:5]):
 plt.subplot(1, 5, plotIndex + 1)
 plt.imshow(np.reshape(test_img[badIndex], (28,28)),
cmap=plt.cm.gray)
 plt.title('Predicted: {}, Actual:
{}'.format(predictions[badIndex], test_lbl[badIndex]),
fontsize = 15)
```



Showing Misclassified Digits

Closing Thoughts

The important thing to note here is that making a machine learning model in scikit-learn is not a lot of work. I hope this post helps you with whatever you are working on. My next machine learning tutorial goes over PCA using Python. If you have any questions or thoughts on the tutorial, feel free to reach out in the comments below, through YouTube video page, or through Twitter!