Python For Applied Machine Learning - Practical 5

Michael Halstead

May 15, 2024

1 Introduction

Today we will be covering:

- Logistic Regression;
- Saving models as pickles; and
- Loading models as pickles then segmenting an input image.

We are going to change up how we set up our code in this practical to get you used to how you will need to do it in the assignment. Instead of having everything in a single script we will further abstract our code into functions. At the bottom of your python file you will have the following code:

```
if __name__=="__main__":
  # parse the command line arguments
  flags = parse_args() # you will need to create the parse_args
    function
  # complete the exercises
  if flags.exr == 'exr0':
    exr0(flags) # you will need to create the exr0 function
  if flags.exr == 'exr1':
    exr1(flags) # you will need to create the exr1 function
```

This is the basics of setting your python script to be used in a functional manner. You will need to create three functions, parse_args(), exr0, and exr1. Your parse_args function can be in your main script but your other two can either be in your main script or a module file (preferred). Your parse_args function should be above your *if __name__=="__main__"* code. An example of what your parse_args function might look like (don't forget to return the flags):

```
def parse_args():
  parser = argparse.ArgumentParser( description='Extracting command
     line arguments', add_help=True )
  parser.add_argument( '--exr', action='store', required=True )
  parser.add_argument( '--dataset', action='store', default='data/
    colour_snippets' )
  parser.add_argument( '--classes', '-c', nargs="+", default=[] )
```

```
6   parser.add_argument( '--split', action='store', type=float,
      default=0.3 )
7   parser.add_argument( '--imax', action='store', type=int, default
      =10 )
8   parser.add_argument( '--logreg', action='store', default='
      logreg_model.pkl' )
9   parser.add_argument( '--image', action='store', default='data/
      sweetpepper.png' )
10  return parser.parse_args()
```

The final two functions will be based on the next two sections.

2 Logistic Regression

In the lecture, you have seen what logistic regression is and how we calculate it. In this practical we will do this for multiple dimensions instead of just 1. We will also use gradient descent to calculate our weights and bias' iteratively, with the aim of getting better representations. We will only be performing logistic regression for two classes, and our metrics will be accuracy score, confusion matrix, and precision recall curve.

The first thing we need to do is code up the logistic regression class, which needs to be in a module file. This class will have four methods: *__init__*, *sigmoid*, *fit*, and *predict*.

The *init* method will take two keyword arguments: learning rate (0.001), and max iterations (100), which will be stored as members of the class. You will also create two other members in this method which will be set to *None*: weights and biases. You can call these four members what ever you like.

The *sigmoid* method will take an X matrix of the shape (samples, features) and perform,

$$\frac{1}{1+e^{-x}} \tag{1}$$

where x is each element in X. You should be able to complete this without a for loop that iterates over X.

For the *fit* function we need matrix X as the training data, and vector y as the labels for X. The first step here is to get the number of samples and the number of features from X. Then we will set the weights to a vector of zeros as the same length of the features in X, and set the bias to 0. Then create a for loop to the maximum number of iterations you stored in your *init* as a member.

In the for loop the calculate the predictions such that,

$$preds = X.weights + bias \tag{2}$$

where X is your input matrix, and weights and bias are your members. You then need to run these predictions (*preds* in the above equation) through your *sigmoid* method using Equation 1, which will produce your updated *preds*. Then you need to calculate the derivatives,

$$dw = \frac{1}{samples} \times X^T.(preds - labels), \tag{3}$$

$$db = \frac{1}{samples} \times \sum preds - labels \tag{4}$$

where samples is the number of samples in X and labels are the input y to the *fit* method. Finally, for this method, you need to iteratively update the weights and bias such that:

$$weights = weights - learningrate \times dw \tag{5}$$

$$bias = bias - learningrate \times db \tag{6}$$

The final method, *predict*, is relatively simple that takes as input a matrix X of the shape (samples, features). The first step is to run X through Equation 2 then put these outputs into your *sigmoid* method. Finally, for all members in the outputs from the *sigmoid* method you need to classify them, if the value is smaller than or equal to 0.5 we classify it as a 0, otherwise it's a 1. You should return the probabilities (outputs from *sigmoid*) and the classification scores.

Now for your *exr0* function. First read in the files from the colour snippets dataset and split them into a training and an evaluation set, make sure you return the appropriate labels. Next we will extract the feature vectors for both of these, the pseudo-code is as follows:

```
def read_files_to_matrix(files, labels):
  for all files
    read in RGB image
    convert RGB image to Lab colour space
    create a feature matrix of size (N,3) for RGB values
    create a feature matrix of size (N,3) for Lab values
    Create labels for all N in the feature matrix
  return the feature matrix and the labels as numpy arrays

Get the training feature matrix and labels
Get the evaluation feature matrix and labels
```

At this point you can choose to do feature normalisation of the training and evaluation features based on the mean and standard deviation of the training set. Then, create two logistic regression objects (one for RGB and one for Lab) with a value of 1 for the max iterations. Fit the two objects using the appropriate data for each. Output the following metrics using your predict method: accuracy score, confusion matrix, and precision recall curves for both the RGB and Lab objects. What do you notice? Play around with different colours to see which ones get confused the most.

2.1 Saving a Pickle

This is an extension to the first section, here we will save the important information we need to reproduce your models without having to run your training

phase again. For this method you might think it's a bit redundant as it's relatively quick to train. But, depending on which methods you choose for your assignment it may take a long time to train a model, I don't want to have to wait hours for your code to train so please use this method to save the important information. This allows me to just load your pickle file and run your code.

```
dit = {}
dit['model'] = <your best logistic regression object there>
dit['mu'] = <your mean for normalisation if used>
dit['std'] = <your standard deviation for normalisation if used>
dit['thresh'] = <The best threshold as calculated by your precision
    recall curve>
with open(<your file name here>, 'wb') as fid:
    pickle.dump(dit, fid)
```

And now I should be able to load your pickle and perform a task based on this model, without having to retrain things.

3 Loading a Pickle and Segmenting an Image

In this section we will load your pickle and segment the image sweetpepper.png which is included in this weeks practical folder. The following is the pseudo-code to complete:

```
with open(<your file name>, 'rb') as fid:
  data = pickle.load(fid)

read in the image sweetpepper.png
get the rows and columns shape of the image
reshape the input image to be size (N,3)
use your model from data to predict your feature matrix, this
    returns the pixel classification and pixel probabilities.

Use the pixel classification and reshape to rows and columns of the
    original image.
Save this reshaped image.

Using the pixel probabilities, if the pixel is less than or equal
    to the threshold in data set this to 1 else it's zero.
Reshape to original image dimensions.
Save this reshaped image.
```

Once you have done this, compare the results. What do you think of the two results? Is there a better way to display this?