

Simple Machine Learning Model

Killian Jochen McShane

I chose to implement a k-nearest-neighbour learning algorithm for my assignment.

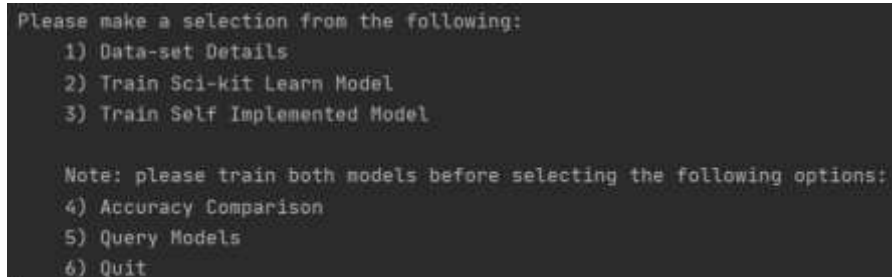
Below is a check list of the marking points. Each has been implemented successfully to the best of my ability.

Running the Program

This section will cover how to run the program, its software dependencies, and explanations of each function's implementation. Details of the parameters and variables will also be included.

Step 1: Navigating the Menu

The user is given a choice of six options to choose from. Below is a screenshot of this menu selection. Both models must be trained before selecting options four and five. This is due to the nature of my self-implemented k-nearest-neighbour algorithm. It was not possible to create a Pickle file for this single model. Options 4 and 5 required the parameters created during the training of this model to display accuracy. Therefore, unfortunately this caveat had to be added.



```
Please make a selection from the following:
1) Data-set Details
2) Train Sci-kit Learn Model
3) Train Self Implemented Model

Note: please train both models before selecting the following options:
4) Accuracy Comparison
5) Query Models
6) Quit
```

Step 2: Data-set Details

Option 1 displays the details of the data-set. The most salient thing to note here is that the data-set has been reduced to 500 samples to help with computation times.



```
Optical Recognition of Handwritten Digits Data-set
-----

Classes: 10 (where each class refers to a digit, 0 to 9.)
Total Samples: 1797 (this has been reduced to 500 to help with computation speed.)
Attributes: 64 pixels per sample.
Samples Per Class: ~100.
Features: 8x8 images of pixels in the integer range 0 to 16.
Instances: 5000.
Train/Test Split: 70%/30%.
```

Step 3: Training Models

Option 2 and 3 allow the user to train both the sci-kit learn model and my own self implemented model of the k-nearest-neighbour algorithm. This must be done before the following options are selected for the reasons given in step 1.

Step 4: Accuracy Comparison

Option 4 allows the user to compare the accuracies of each model's training and testing sets. As you can see from the image, the accuracies are quite similar.

```
Sci-kit Learn Model
-----
Training Accuracy: 0.99
Testing Accuracy: 0.98
Percentage Difference: 0.86 %

Self Implemented Model
-----
Training Accuracy: 0.98
Testing Accuracy: 0.99
Percentage Difference: -0.71 %
```

Step 5: Querying

Option 5 allows the user to select an index from the testing data-set to query the prediction. The example given below is a correct prediction of digit 4.

```
5
Enter an index from 350 to 499 to query the model on the test data-set:
450
The model predicted the digit as: [4]
Its corresponding real label was: [4]
```

Software Dependencies

Python 3.8 including the numpy, pandas, sklearn, time and collections modules. The latter two are already included in the python environment and do not need to be installed.

Implementation Details of my Algorithm

The basic idea of the algorithm is to first generate training and testing sets similar to how the test_train_split function of sklearn operates.

Then a simple function determines the Euclidean distance from one digit to another. This function is called distances() and takes the parameters of two digits.

```

elif selection.lower() == '3' or selection.lower() == 'self implemented model':
    '''
    The following is my own implementation of a knn algorithm.
    The random module is used to mirror sklearn's test_train_split function.
    I chose to use a seed (1997) to allow for replication of the results.
    The slice-notation is used to select from the list of indices easily
    I chose to use a training/testing split of 70%/30%.
    '''
    np.random.seed(1997)
    indices = np.random.permutation(len(df))
    '350 is 70% of the total 500 samples used.'
    n_training_samples = 350

    X_train = digits['data'][indices[:-n_training_samples]]
    X_test = digits['data'][indices[-n_training_samples:]]
    y_train = digits['target'][indices[:-n_training_samples]]
    y_test = digits['target'][indices[-n_training_samples:]]

    'The distances function determines the euclidean distance between two digits'
    def distances(digit1, digit2):
        digit1 = np.array(digit1)
        digit2 = np.array(digit2)

        return np.linalg.norm(digit1 - digit2)

```

The function neighbours() takes the parameters of the training data, the training labels, the selected digit and the specified number of k nearest neighbours. This function calls the distances function and adds each distance to a list called distance_list.

The distance list is sorted and the first k distances are selected to a new list called neighbour_list.

A function called vote() takes this neighbour_list as a parameter and determines a single result for the prediction calculation.

```

'''
The neighbours function calculates the distance of a digit from all other digits in the set.
It then appends these distances to the distance_list list. This list is then sorted using
a lambda function (x: x[1]). Only the k nearest neighbours are then added to the neighbour_list.
'''
def neighbours(X_train, y, digit, k):
    distance_list = []
    for i in range(len(X_train)):
        digit_distance = distances(digit, X_train[i])
        distance_list.append((X_train[i], digit_distance, y[i]))
        distance_list.sort(key=lambda x: x[1])
        neighbour_list = distance_list[:k]

    return neighbour_list

'The vote function votes to get a single result.'
def vote(neighbour_list):
    digit_counter = Counter()
    for digit in neighbour_list:
        digit_counter[digit[2]] += 1
    result = digit_counter.most_common(1)[0][0]
    return result

```

Finally the accuracy of both the training set and the test set are determined via their respective functions, `get_test_accuracy()` and `get_train_accuracy()` by simply dividing the correct number of results over the total number of each set.

```

'Simple function to calculate correctly predicted test results divided by the total number.'
def get_test_accuracy():
    correct_test = 0
    total_test = 0
    for i in range(500-n_training_samples):
        neighbour_list = neighbours(X_train, y_train, X_test[i], 2)
        if y_test[i] == vote(neighbour_list):
            correct_test += 1
            total_test += 1
    self_implemented_test_accuracy = float('%2f' % (correct_test/total_test))

    return self_implemented_test_accuracy

'Simple function to calculate correctly predicted training results divided by the total number.'
def get_train_accuracy():
    correct_train = 0
    total_train = 0
    for i in range(n_training_samples):
        neighbour_list = neighbours(X_train, y_train, X_test[i], 2)
        if y_test[i] == vote(neighbour_list):
            correct_train += 1
            total_train += 1
    self_implemented_train_accuracy = correct_train/total_train

    return self_implemented_train_accuracy

```