CSC 4780/6780
Fall 2022
Homework 09

October 16, 2022

This homework is due at 11:59 pm on Sunday, Oct 23. It must be uploaded to iCollege by then. No credit will be given for late submissions. A solution will be released by noon on Monday, Oct 24.

it is always a good idea to get this done and turned in early. You can turn it in as many times as you like – iCollege will only keep the last submission. If, for some reason, you are unable to upload your solution, email it to me before the deadline.

For this homework, in particular, `digit_test.py` will take a **very long time** to run. (On my Mac laptop, it took nearly four hours to complete.) If you finish it before Sunday, you can run it overnight and there will be no stress. If you finish it on Sunday afternoon, you may not make the deadline.

Incidentally, I rarely check my iCollege mail, but I check my `dhillegass@gsu.edu` email all the time. Send messages there.

1 Recognizing MNIST digits using k-nearest neighbor

The classifiers we've used so far can divide the data space into relatively simple shapes. The k-nearest neighbor can deal with very complex shapes. One of the challenges of doing k-nearest neighbor is that it is pretty slow and takes up a lot of memory.

The National Institute of Standards had high school students write the digits 0-9. These were scanned into 28 by 28 black and white images. This data set is known as MNIST, and it is a really common machine learning task.

There are 60,000 training images and 10,000 testing images.

1.1 Getting the data

The keras deep learning library makes it really easy to download the MNIST dataset. Install Tensorflow:

```
pip3 install tensorflow
```

(If you are on a M1 Mac, I think you will need to do `pip3 install tensorflow-macos` instead.)

Install Keras:

```
pip3 install keras
```

For this exercise, we are going to reshape each image to be a 784 dimension vector instead of a 28 by 28 matrix:

```
from keras.datasets import mnist

d = 28 * 28
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape((-1, d))
X_test = X_test.reshape((-1, d))
```

(Of course, because we are creating two different programs `digit_train.py` and `digit_test.py`, you'll only ever need one. For example, in `digit_train.py` it will look like this:

```
d = 28 * 28
(X_train, y_train), (_, _) = mnist.load_data()
X_train = X_train.reshape((-1, d))
n = X_train.shape[0]
```

)

1.2 Train a k-nearest neighbor classifier using scikit-learn

You are going to use grid cross validation to find the best values for k, the distance metric, and whether it weighs the results by distance.

Create a program called `digit_train.py` that

- Reads in the MNIST training data.

- Creates a dictionary of the hyperparameters you would like to try:
  - `metric:["euclidean", "manhattan"]`
  - `n_neighbors":  [1,3,5,7]`
  - `weights":['uniform', "distance"]`
- Create a `KNeighborsClassifier` with no parameters.
- Create a `GridSearchCV` in verbose mode (3) and 4-fold cross validation.
- Run the cross validation.
- Print how long it took.
- Print each combination and its accuracy from most accurate to least.
- Create a new `KNeighborsClassifier` with the best possible parameters.
- Fill it with the training data.
- Write the classifier to `knn_model.pkl`

Save the output as `digit.txt`.

There is a discussion oif `GridSearchCV` in chapter 14 (Optimizing Models Using AutoML).

```
> python3 digit_train.py > digit.txt
> cat digit.txt
Fitting 4 folds for each of 16 candidates, totalling 64 fits
[CV 1/4] END metric=euclidean, n_neighbors=1, weights=uniform;, score=0.969 total time=  11.7s
[CV 2/4] END metric=euclidean, n_neighbors=1, weights=uniform;, score=0.969 total time=  11.7s
...
[CV 3/4] END metric=manhattan, n_neighbors=7, weights=distance;, score=0.960 total time= 6.9min
[CV 4/4] END metric=manhattan, n_neighbors=7, weights=distance;, score=0.962 total time= 6.9min
Took 13643.5 seconds
metric=euclidean n_neighbors=3 weights=distance : Mean accuracy=97.05%
metric=euclidean n_neighbors=5 weights=distance : Mean accuracy=96.97%
...
metric=manhattan n_neighbors=7 weights=uniform : Mean accuracy=96.05%
Best parameters: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'distance'}
Params: {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'euclidean', 'metric_params': None,
                    'n_jobs': None, 'n_neighbors': 3, 'p': 2, 'weights': 'distance'}
Accuracy on training data = 100.0%
Wrote knn_model.pkl
```

1.3 Test a k-nearest neighbor classifier using scikit-learn

Create a program called `digit_test.py` that:

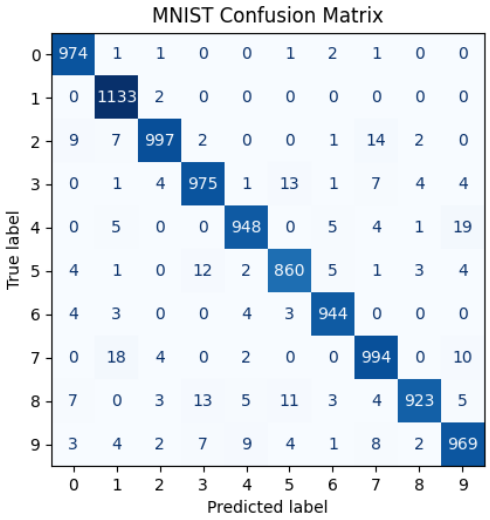
- Loads the testing data for MNIST
- Loads a `KNeighborsClassifier` from `knn_model.pkl`
- Does predictions for X_test.

- Prints the amount of time that the inference takes.
- Prints the accuracy of those predictions.
- Creates a confusion matrix plot and saves it as `sk_confusion.png`

When mine runs, it looks like this:

```
> python3 digit_test.py
Inference: elapsed time = 12.11 seconds
Accuracy on test data = 97.2%
Confusion:
[[ 974    1    1    0    0    1    2    1    0    0]
 [   0 1133    2    0    0    0    0    0    0    0]
 [   9    7  997    2    0    0    1   14    2    0]
 [   0    1    4  975    1   13    1    7    4    4]
 [   0    5    0    0  948    0    5    4    1   19]
 [   4    1    0   12    2  860    5    1    3    4]
 [   4    3    0    0    4    3  944    0    0    0]
 [   0   18    4    0    2    0    0  994    0   10]
 [   7    0    3   13    5   11    3    4  923    5]
 [   3    4    2    7    9    4    1    8    2  969]]
Wrote sk_confusion.png
```

And `sk_confusion.png` looks like this:



2 6780 Students only: Learn to use Faiss

In general, using faiss instead of Sci-kit Learn will decrease the time for inference by a factor of 10. If you use approximate KNN, that will decrease your time by another factor of 10. If you do the

processing on a GPU, that will decrease your time by another factor of 10.

So, you can do inference about 1000 times faster if you use faiss wisely.

Install faiss on your machine. You have a newish NVIDIA graphics card? The default faiss will use it:

```
> pip3 install faiss
```

You don't? faiss can also run on your CPU:

```
> pip3 install faiss-cpu
```

2.1 faiss_train.py

Create a python program called `faiss_train.py` that takes one argument: `exact` or `approximate`. This program should:

- Read in the MNIST training data.
- Create a faiss index object
- Train it on the data.
- Write the index to `exact.faiss` or `approx.faiss` depending on the argument.
- Write the classes that are being indexed (that is `y_train`) to `values.pkl`

For exact, the index should be created like this:

```
index = faiss.IndexFlatL2(d)
```

For approximate, the index should be created like this:

```
quantizer = faiss.IndexHNSWFlat(d, 32)
index = faiss.IndexIVFPQ(quantizer, d, 1024, 16, 8)
```

To train it, you can train and then add with the training data:

```
index.train(X_train)
index.add(X_train)
```

For these two methods to work `X_train` needs to be a numpy array of type Float32. You can change the type of a numpy array like this:

```
X_train = X_train.astype("float32")
```

Writing an index is done with `faiss.write_index`:

```
faiss.write_index(index, path)
```

When mine runs, it looks like this:

```
> python3 faiss_train.py exact
Using IndexFlatL2 for true KNN
Index has 60000 data points.
Wrote index to values.pkl and exact.faiss
```

Or

```
> python3 faiss_train.py approximate
Using HNSW/IVFPQ for approximate KNN
Index has 60000 data points.
Wrote index to values.pkl and approx.faiss
```

2.2 faiss_test.py

Create a python program called `faiss_test.py` that takes one argument: `exact` or `approximate`. This program should:

- Read in the MNIST testing data.
- Read in the appropriate faiss index and `values.pkl`
- Does predictions (using uniform, not weighted) for the MNIST testing data
- Prints the time consumed by inference
- Prints the accuracy
- Makes a confusion matrix plot and saves it to `exact_confusion.png` or `approx_confusion.,png`

Mine looks like this when it runs:

```
> python3 faiss_test.py exact
Read exact.faiss: Index has 60000 data points.
Inference: elapsed time = 2.34 seconds
Accuracy on test data = 97.0%
Confusion:
```

```
[[ 974    1    1    0    0    1    2    1    0    0]
 [   0 1133    2    0    0    0    0    0    0    0]
 [  10    9  996    2    0    0    0   13    2    0]
 [   0    2    4  976    1   13    1    7    3    3]
 [   1    6    0    0  950    0    4    2    0   19]
 [   6    1    0   11    2  859    5    1    3    4]
 [   5    3    0    0    3    3  944    0    0    0]
 [   0   21    5    0    1    0    0  991    0   10]
 [   8    2    4   16    8   11    3    4  914    4]
 [   4    5    2    8    9    2    1    8    2  968]]
Wrote exact_confusion.png
```

Or

```
> python3 faiss_test.py approx
Read approx.faiss: Index has 60000 data points.
Inference: elapsed time = 0.16 seconds
Accuracy on test data = 95.4%
Confusion:
[[ 968    1    3    0    0    2    5    0    1    0]
 [   1 1127    4    0    1    0    2    0    0    0]
 [   9    7  984    7    1    0    2   13    8    1]
 [   0    0    6  948    1   20    0    6   24    5]
 [   1    7    0    0  923    0    7    7    1   36]
 [   3    0    0   26    3  837   10    2    6    5]
 [   6    3    3    1    1    6  936    0    2    0]
 [   0   17    9    0   11    1    0  970    0   20]
 [   4    1    4   17    4   24    4    7  902    7]
 [   3    2    2    6   17    5    1   23    8  942]]
Wrote approx_confusion.png
```

Note that unlike Scikit-Learn, faiss doesn't give you the final predictions. It just gives you the index of the training data rows that are closest to the input. and the distances. So, your python challenge is to make predictions out of it. I used the following numpy methods:

- `take`
- `bincount`
- `argmax`
- `apply_along_axis`

We are not weighting by distance, so you can ignore the distances.

My `faiss_test.py` has no loops at all. Good luck!

3 Criteria for success

If your name is Fred Jones, you will turn in a zip file called `HW08_Jones_Fred.zip` of a directory called `HW08_Jones_Fred`. It will contain:

- `digit_train.py`
- `digit_test.py`
- `sk_confusion.png`
- `digit.txt`

If you are in 6780, also include the following in your zipfile:

- `faiss_train.py`
- `faiss_test.py`
- `exact_confusuion.png`
- `approx_confusion.png`

Be sure to format your python code with black before you submit it.

The other files (like `knn_model.pkl` are very big. Please do not include anything not listed here.

We would run your code like this:

```
cd HW09_Jones_Fred
python3 digit_train.py
python3 digit_test.py
```

(But, of course, it will take too long to run everyone's this way. We will probably just look at `digit.txt` and your code.)

If you are in 6780, we will run your code like this:

```
cd HW09_Jones_Fred
python3 faiss_train.py exact
python3 faiss_train.py approximate
python3 faiss_test.py exact
python3 faiss_test.py approximate
```

Do this work by yourself. Stackoverflow is OK. A hint from another student is OK. Looking at another student's code is *not* OK.

The template files for the python programs have import statements. Do not use any frameworks not in those import statements.

4 Reading

You should have read through "Using Bayesian Search" in chapter 14 ("Optimizing Models and Using AutoML").

Here is a video on K-Nearest Neighbor: `https://youtu.be/HVXime0nQeI`

Here is a pretty, good video on decision trees: `https://youtu.be/ZVR2Way4nwQ` That's next!