1 *Lab: Digit recognition*

For this lab, some of the computations take a lot of time. For the most time-intensive computations, you are instructed to omit the code from your stencil so submitting doesn't take forever!

In this lab, you will test out some methods for recognizing handwritten digits. The training data consist of a bunch of 28×28 images of handwritten digits. For each image, you are also given the correct digit (the *label*).[1]

Given a new input image, your program should guess the digit it depicts by finding the nearest image in the training data and outputting the label of the closest image. This approach to learning is called *nearest neighbor*. You will evaluate the approach by using it to guess the digits depicted by a bunch of test images (that come with labels) and reporting the error rate.

What do we mean by nearest? How do we measure the distance between two images? The images are represented as Vecs, so it is natural to use the norm (or squared norm) of the difference between the two vectors.

Computing the squared norm involves squaring and adding up $28^2 = 784$ numbers. Comparing the input image to 3,000 training images thus involves computing $3000 \cdot 784 = 2,352,000$ squares.

An appealing alternative is to choose a small number of vectors $\boldsymbol{v}_1, \ldots, v_k$ (perhaps k is ten or twenty), and precompute for each training vector $\boldsymbol{v}$ the coordinate representation of $\boldsymbol{v}$ in terms of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$. For each input image $\boldsymbol{u}$, compute *its* coordinate representation, and compare it to the coordinate representations of the training vectors; output the label of the nearest coordinate representation. This has several advantages:

1. It takes much less computation to find the nearest coordinate representation in terms of twenty vectors; comparing the coordinate representation of the input image to the coordinate representations of 3,000 training images involves computing $3000 \cdot 20 = 60,000$ squares.

2. If $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ are orthonormal, computing the coordinate representation is quick and easy.

3. If $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ are orthonormal, the distance between coordinate representations of two vectors equals the distance between the two vectors.

However, it is highly unlikely that such a small number of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ can span all of the training data. No matter which k vectors you choose, most of the training images will not *have* coordinate representations in terms of those vectors. Instead, each image is projected onto the span of those vectors, and we use the coordinate representation of the *projection*. This approach preserves the first two advantages:

1. It takes much less computation to find the nearest coordinate representation.

2. If $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ are orthonormal, computing the coordinate representation of the projection onto Span $\{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k\}$ is quick and easy.

[1] The data we use is from the *MNIST Database of handwritten digits* NIST is the US's National Institute of Standards and Technology. This institute provided the original images, which were processed by Yann LeCun. The files we provide are derived from files he created.

What about the third advantage? The distance between coordinate representations of the projections of two vectors is the same as the distance between the projections, but the distance between the projections of two vectors is not likely to be the same as the distance between the two vectors themselves. This suggests that using nearest neighbor on the images' projections (or their coordinate representations) is likely to have a worse error rate than using nearest neighbor on the original images. If the error rate is not *too* much worse, we might consider it worth using projection because of the time savings.

If the vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k$ are chosen randomly, the error rate becomes much worse. The surprise is that, if they are chosen to be principal components (right singular vectors), the error rate is actually *better* than the error rate for the images themselves. Using this method yields both better accuracy and better computational performance.

(Another fun fact: every task in this lab can be achieved using a one-liner, including each of the procedures.)

Task 1.1: Write the procedure `sq_dist(u, v)` that, given two Vecs, returns the squared distance between them.

Task 1.2: Write the procedure `nn(u, veclist)` that, given a Vec `u` and a list `veclist` of Vecs, returns the index i of the Vec in `veclist` whose squared distance from `u` is the smallest. Your procedure should use `sq_dist(u, v)`.

Task 1.3: Write the procedure `nn_label(u, veclist, labels)` that, given a Vec `u`, a list `veclist` of Vecs and a list `labels` of corresponding labels, returns the label of the element of `veclist` nearest to `u`. Your procedure should use `nn(u, veclist)`.

Task 1.4: Write the procedure `error_rate(guessed_labels, correct_labels)` that, given two equal-length lists of labels, reports the percentage of corresponding labels that do not match.

We provide files `mnist-images.dat` and `mnist-labels.dat` with binary representations of 3,100 images and the corresponding labels, and a module, `mnist_loader`, that defines a procedure `load_data(n)` that returns a pair of lists of length n, a list of Vecs, each representing an image, and a list of the corresponding labels.

Use the procedure to load 3100 images and the corresponding labels.

```
>>> from mnist_loader import load_data
>>> images, labels = load_data()
```

Use the `image` module (provided to you) to see what the images look like:

```
>>> v = images[0]
>>> import image
>>> image.image2display([[v[i,j] for j in range(28)] for i in range(28)])
```

The first 3000 images and corresponding labels will be used for training. The 100 images and labels after those will be used for testing.

```
>>> train_images = images[:3000]
>>> train_labels = labels[:3000]
>>> test_images = images[3000:3100]
>>> test_labels = labels[3000:3100]
```

Next you will try out basic nearest-neighbor (without any projection). Use the Python REPL.

Task 1.5: Assign to `guessed_labels` the list of labels that nearest-neighbor assigns to images in the 100-element list `test_images`.

Note: This takes about six seconds per image. I recommend you first try the computation on a 10-element list, `test_images[:10]`. The labels I got were:

```
[3, 5, 1, 4, 1, 9, 7, 7, 0, 5]
```

Compare those guessed labels to the true labels, `test_labels[:10]`.

Once you've verified that you've got the right approach, start the process of computing the list of guessed labels for all 100 images. You can copy and paste the list into your stencil.

When you are ready to compute the labels for the 100-element lists, I suggest you time the computation. Here's an easy way to do that. First, import the `time` module into your Python REPL. Next, open a scratch file with your text editor, and put in the following code.

```
t = time.perf_counter()
(your code here)
print("Number of seconds: ",  time.perf_counter() - t)
```

Copy those three lines, and paste into your Python REPL.

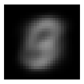
While you're waiting for the computation to complete, you can open a second Python session and skip ahead to Task 1.7.

Task 1.6: Apply the procedure `error_rate(guessed_labels,correct_labels)` to find the error rate of the nearest-neighbor algorithm when trained on the first 3000 images and tested on the last hundred. Copy and paste the result into your stencil.

Next you will center the training data (translating by subtracting the centroid) and find an orthonormal basis for the 20-dimensional vector space nearest to the centered training images.

Task 1.7: Write the procedure `find_centroid(veclist)` that, given a list of Vecs, returns their centroid.

Task 1.8: Use your `find_centroid(veclist)` procedure to find the centroid of the training images.

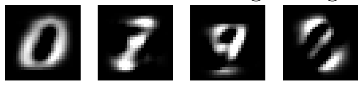
Here's how the centroid looks: 

Task 1.9: Find the list of centered training images by subtracting the centroid from each of the training images.

Task 1.10: Translate the test images in the same way, subtracting from each the centroid of the *training* images.

We provide a module `power_svd` that defines a procedure `right_singular_vectors(A, k)` to return rough approximations[2] of the first k right singular vectors of a Mat A.

Task 1.11: Find the first twenty right singular vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{20}$ of the matrix whose rows are the centered training images.

Here's how the first few right singular vectors look (with entries scaled up to appropriate pixel intensity scale): 

First you'll find the coordinate representations of the 3,000 training images in terms of just the first ten right singular vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{10}$ and use these in nearest-neighbor search on the hundred test images. Then you'll try the same experiment but using the first twenty right singular vectors. Finally, you'll try the same experiment but using twenty random orthonormal vectors.

First you will work in $\mathcal{V}_{10}$, the span of the first ten right singular vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{10}$.

Task 1.12: To help you get the coordinate representations of the projections onto $\mathcal{V}_{10}$, construct the matrix `M10` whose rows are the first ten right singular vectors.

Multiplying an image by `M10` yields the coordinate representation of the projection of the image onto $\mathcal{V}_{10}$.

Task 1.13: Apply nearest neighbors to the coordinate representations. This involves (1) finding the coordinate representations of the projections of the training images and of the test images, and (2) using `nn_label(v, veclist, labels)` to guess a label for each of the projections of the test images. Assign to `guessed_labels_10` the list of labels that nearest-neighbor assigns to images in the 100-element list `test_images`.

Note: Just as you did for for Task 1.5, do this computation in the Python REPL, not in your stencil. You wouldn't want to repeat the computation every time you run the submit script! As before, try it out first on a small number of test images before trying it on all 100 test images. Here's what I got for the first ten test images:

```
[3, 5, 1, 4, 6, 4, 7, 7, 5, 5]
```

Compare those guessed labels to the true labels, `test_labels[:10]`.

As before, I recommend timing the computation using the `time` module.

[2] Really rough. The corresponding singular value estimates aren't even guaranteed to be in nonincreasing order.

Task 1.14: Find the error rate for nearest neighbor applied to the coordinate representations of the projections onto $\mathcal{V}_{10}$.

Next you will work in $\mathcal{V}_{20}$, the span of the first twenty right singular vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{20}$.

Task 1.15: To help you get the coordinate representations of the projections onto $\mathcal{V}_{20}$, construct the matrix `M20` whose rows are the first twenty right singular vectors.

Task 1.16: Apply nearest neighbors to the coordinate representations. This involves (1) finding the coordinate representations of the projections of the training images and of the test images, and (2) using `nn_label(v, veclist, labels)` to guess a label for each of the projections of the test images. Assign to `guessed_labels_20` the list of labels that nearest-neighbor assigns to images in the 100-element list `test_images`.

Note: Just as you did for for Task 1.5, do this computation in the Python REPL, not in your stencil. You wouldn't want to repeat the computation every time you run the submit script! As before, try it out first on a small number of test images before trying it on all 100 test images. Here's what I got for the first ten test images:

```
[3, 5, 1, 4, 1, 9, 7, 7, 8, 5]
```

Compare those guessed labels to the true labels, `test_labels[:10]`.

Task 1.17: Find the error rate of nearest neighbor applied to the coordinate representations of the projections onto $\mathcal{V}_{20}$.

Is the success of projecting onto $\mathcal{V}_{20}$ a result of the right choice of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{20}$, or would any orthonormal vectors work just as well?

Task 1.18: *Optional:* Generate ten random vectors with the same domains as the image vectors. To generate the entries, import the module `random` and use `random.gauss(0,1)`.

Task 1.19: *Optional:* Generate a list of orthonormal vectors from the random vectors found in Task 1.18 by using the procedure `orthonormalize`.

Task 1.20: *Optional:* Construct a matrix whose rows are the orthonormal vectors from Task 1.19.

Task 1.21: *Optional:* Use the matrix of Task 1.20 to obtain coordinate representations of the projections of the training and test vectors, and try out nearest neighbor. What was the error rate?

Final remarks: There are better ways to do nearest neighbor than the naive method. For example, there is a data structure called a *k-d tree*. Given a bunch of training vectors, you can build a k-d tree representation. Once this has been built, it can be searched to find the nearest neighbor to a given query vector.

The bad news is that the k-d tree works pretty badly on the raw image vectors. (This is not hard to understand once you understand k-d trees.) The good news is that it works wonderfully on the coordinate representations in terms of twenty or so right singular vectors. This is related to the fact that the directions of these vectors (the principal components) are chosen so as to maximize the spreading-out of the projections of the training examples along the principal components.