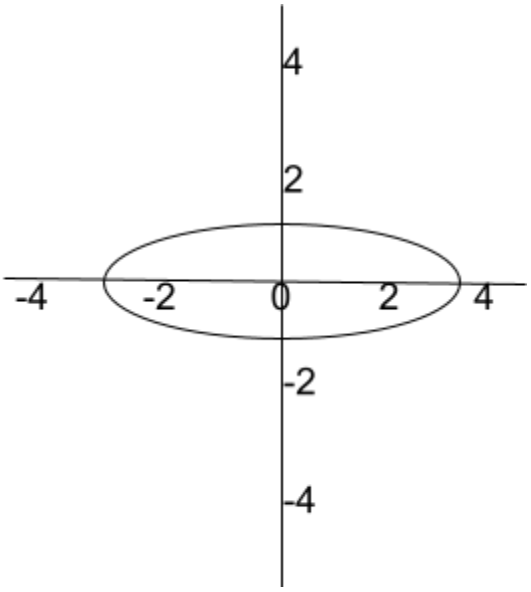
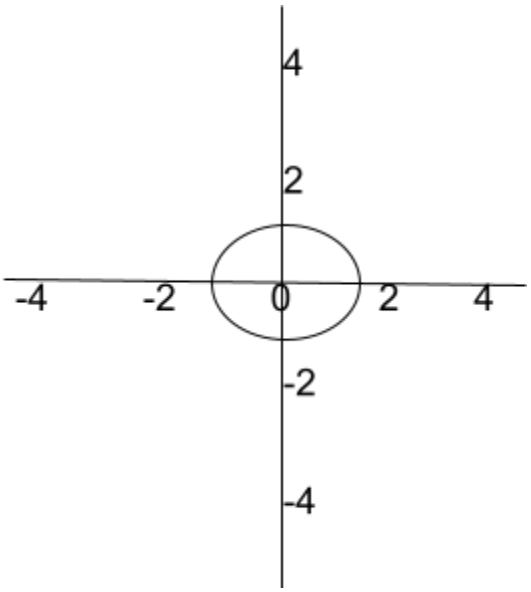
1.

a.



b.



2.

a. q is negative, p and r just need to be non-negative

b. q is 0, p and r just need to be non-negative

c. magnitude of q is equal to $\sqrt{r} * \sqrt{p}$, and p and r must be non-negative

d. q is 0, r is 0, p just needs to be non-negative

3. $|x| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$

$u = \left(\frac{1}{\sqrt{14}}, \frac{2}{\sqrt{14}}, \frac{3}{\sqrt{14}}\right)$

4. $\left(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right), \left(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$

5. w = [2 -1 6]

6. A is 10 X 30, B is 30 X 20

7.

a. n X 1

b. n X n

c. $(i, j) = x^i * x^j$

8.

$xx^T = 1 * 1 + 3 * 3 + 5 * 5 = 35$

$x^T x =$

1*1=1	1*3=3	1*5=5
3*1=3	3*3=9	3*5=15
5*1=5	5*3=15	5*5=25

9.

M =

3	1	-2
1	0	0
-2	0	6

10.

a. $|A| = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 = 40320$

b. $a^{-1} =$

1	0	0	0	0	0	0	0
0	1/2	0	0	0	0	0	0
0	0	1/3	0	0	0	0	0
0	0	0	1/4	0	0	0	0
0	0	0	0	1/5	0	0	0
0	0	0	0	0	1/6	0	0
0	0	0	0	0	0	1/7	0
0	0	0	0	0	0	0	1/8

11.

a. UU^T = diag(1, 1, 1, 1, 1, …. 1)

ie. a diagonal matrix where the diagonal is all one and all other entries are zero

b. $U^{-1} = U^T$

12. $det(A) = 0 = (1 * z) - (2 * 3)$

$z - 6 = 0$

$z = 6$

13.

Gaussian generative models for handwritten digit classification

Recall that the 1-NN classifier yielded a 3.09% test error rate on the MNIST data set of handwritten digits. We will now see that a Gaussian generative model does almost as well, while being significantly faster and more compact.

1. Set up notebook and load in data

As usual, we start by importing the required packages and data. For this notebook we will be using the *entire* `MNIST` dataset. The code below defines some helper functions that will load `MNIST` onto your computer.

```
%matplotlib inline
import matplotlib.pyplot as plt
import gzip, os
import numpy as np
import sys
from scipy.stats import multivariate_normal

if sys.version_info[0] == 2:
    from urllib import urlretrieve
else:
    from urllib.request import urlretrieve
```

[2] ✓ 6.3s Python

```python
# Function that downloads a specified MNIST data file from Yann Le Cun's website
def download(filename, source='http://yann.lecun.com/exdb/mnist/'):
    print("Downloading %s" % filename)
    urlretrieve(source + filename, filename)

# Invokes download() if necessary, then reads in images
def load_mnist_images(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset=16)
    data = data.reshape(-1,784)
    return data

def load_mnist_labels(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset=8)
    return data
```

[3] ✓ 0.0s Python

Now load in the training set and test set

```python
## Load the training set
train_data = load_mnist_images('train-images-idx3-ubyte.gz')
train_labels = load_mnist_labels('train-labels-idx1-ubyte.gz')

## Load the testing set
test_data = load_mnist_images('t10k-images-idx3-ubyte.gz')
test_labels = load_mnist_labels('t10k-labels-idx1-ubyte.gz')
```

[4] ✓ 0.4s Python

The function **displaychar** shows a single MNIST digit. To do this, it first has to reshape the 784-dimensional vector into a 28x28 image.

+ Code + Markdown

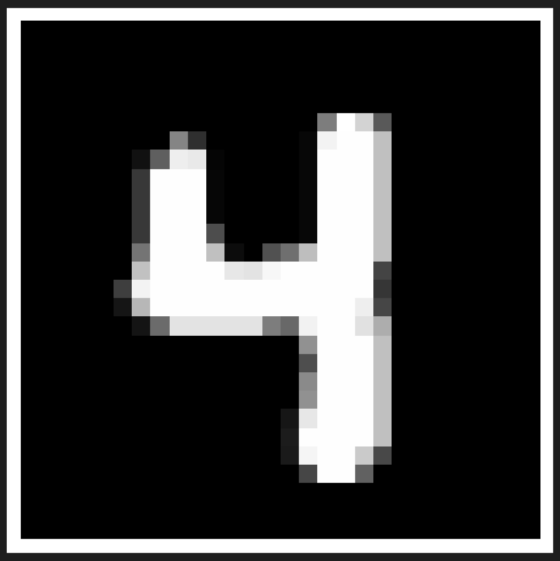
```
def displaychar(image):
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.axis('off')
    plt.show()
```

[5] ✓ 0.0s Python

```
displaychar(train_data[58])
```

[6] ✓ 0.1s Python

...



The training set consists of 60,000 images. Thus `train_data` should be a 60000x784 array while `train_labels` should be 60000x1. Let's check.

```
[7]  train_data.shape, train_labels.shape
     ✓ 0.0s                                                    Python
...  ((60000, 784), (60000,))
```

```
[8]  # split training set into two pieces
     train_data_new = train_data[0:5000, :]
     train_labels_new = train_labels[0:5000]

     val_data = train_data[5000:6000, :]
     val_labels = train_labels[5000:6000]
     ✓ 0.0s                                                    Python
```

```
[9]  train_data_new.shape, train_labels_new.shape
     ✓ 0.0s                                                    Python
...  ((5000, 784), (5000,))
```

```
[10] val_data.shape, val_labels.shape
     ✓ 0.0s                                                    Python
...  ((1000, 784), (1000,))
```

2. Fit a Gaussian generative model to the training data

For you to do: Define a function, **fit_generative_model**, that takes as input a training set (data `x` and labels `y`) and fits a Gaussian generative model to it. It should return the parameters of this generative model; for each label `j = 0,1,...,9`, we have:

- `pi[j]`: the frequency of that label
- `mu[j]`: the 784-dimensional mean vector
- `sigma[j]`: the 784x784 covariance matrix

This means that `pi` is 10x1, `mu` is 10x784, and `sigma` is 10x784x784.

We have already seen how to fit a Gaussian generative model in the Winery example, but now there is an added ingredient. The empirical covariances are very likely to be singular (or close to singular), which means that we won't be able to do calculations with them. Thus it is important to **regularize** these matrices. The standard way of doing this is to add `cI` to them, where `c` is some constant and `I` is the 784-dimensional identity matrix. (To put it another way, we compute the empirical covariances and then increase their diagonal entries by some constant `c`.)

This modification is guaranteed to yield covariance matrices that are non-singular, for any `c > 0`, no matter how small. But this doesn't mean that we should make `c` as small as possible. Indeed, `c` is now a parameter, and by setting it appropriately, we can improve the performance of the model. We will study **regularization** in greater detail over the coming weeks.

Your routine needs to choose a good setting of `c`. Crucially, this needs to be done using the training set alone. So you might try setting aside part of the training set as a validation set, or using some kind of cross-validation.

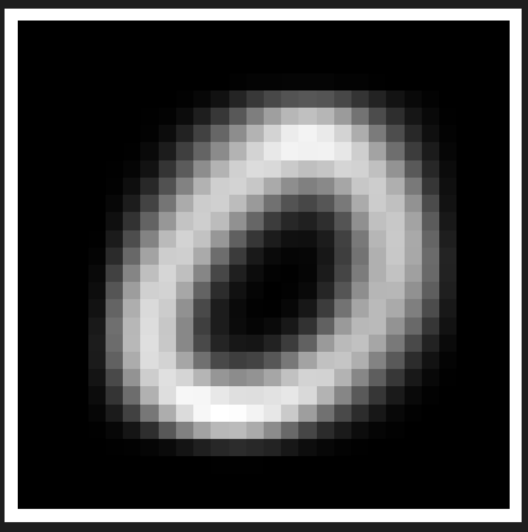
```python
def fit_generative_model(x,y):
    k = 10  # labels 0,1,...,k-1
    d = (x.shape)[1]  # number of features
    mu = np.zeros((k,d))
    sigma = np.zeros((k,d,d))
    pi = np.zeros(k)

    data = [np.zeros(x.shape[1])] * 10
    # finds frequency of labels and does half of mean calculations
    for i in range(len(x)):
        pi[y[i]] += 1
        mu[y[i]] += x[i]

        data[y[i]] = np.row_stack((data[y[i]], x[i]))

    # finishes mean calculations
    for i in range(k):
        mu[i] /= pi[i]

        data[i] = np.delete(data[i], 0, axis=0) # need to delete since first row was filler row of 0s
        data[i] = data[i].T # transposed for use in np.cov function later

    # bgein covariance calculations
    for i in range(k):
        sigma[i] = np.cov(data[i])

    # Halt and return parameters
    return mu, sigma, pi
```

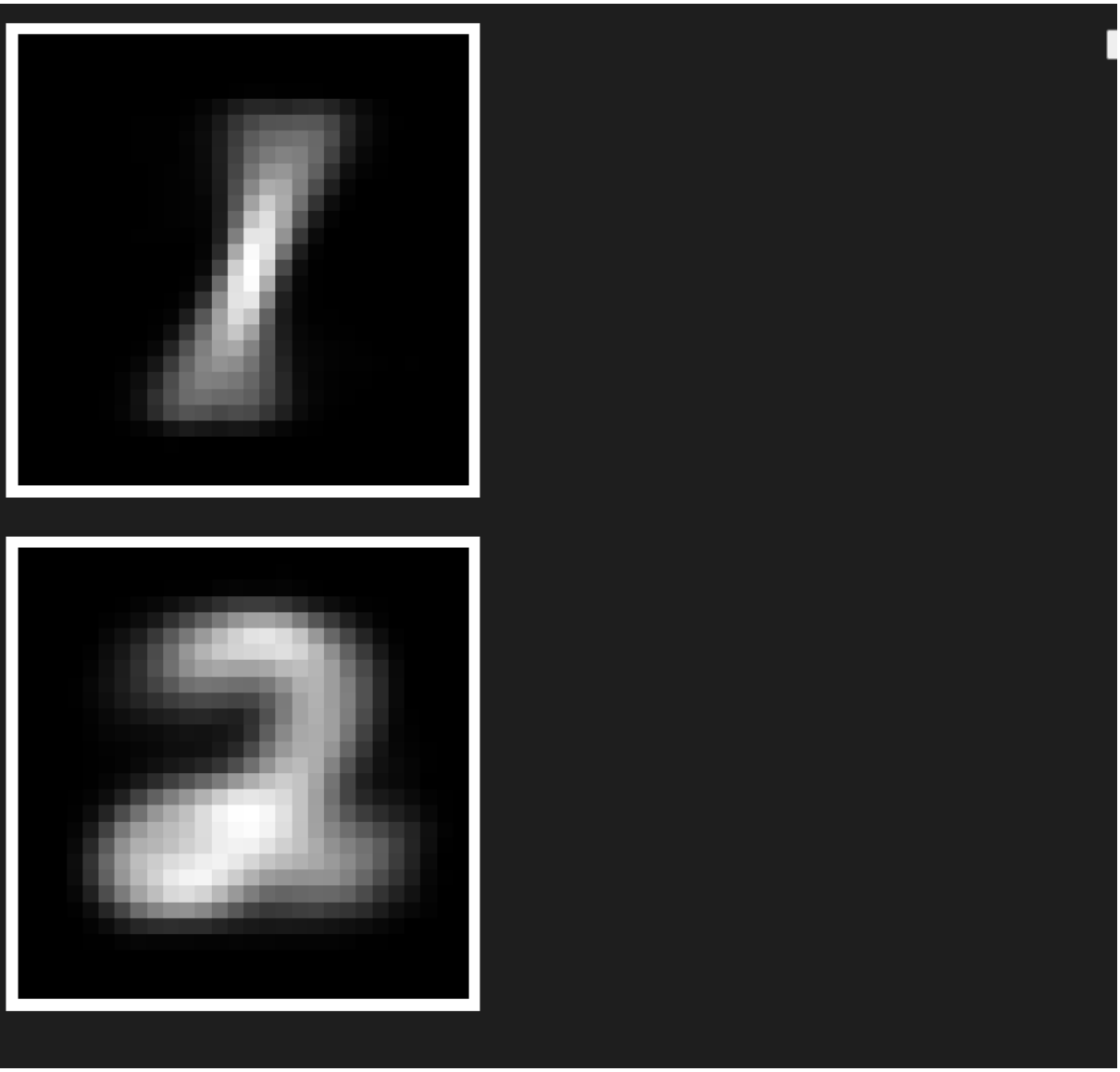
✓ 0.0s Python

Okay, let's try out your function. In particular, we will use **displaychar** to visualize the means of the Gaussians for the first three digits. You can try the other digits on your own.

```python
mu, sigma, pi = fit_generative_model(train_data_new, train_labels_new)
displaychar(mu[0])
displaychar(mu[1])
displaychar(mu[2])
```

✓ 4.9s Python





```python
c = 10000
I = np.identity((train_data.shape)[1] )
reg_sigma = sigma + c * I
```

✓ 0.0s Python

```python
# Compute log Pr(label|image) for each [validation image,label] pair.
# Used to pick optimal c value
k = 10
score = np.zeros((len(val_labels),k))
for label in range(0,k):
    rv = multivariate_normal(mean=mu[label], cov=reg_sigma[label])
    for i in range(0,len(val_labels)):
        score[i,label] = np.log(pi[label]) + rv.logpdf(val_data[i,:])
predictions = np.argmax(score, axis=1)
# Finally, tally up score
errors = np.sum(predictions != val_labels)
print("Your model makes " + str(errors) + " errors out of 1000")
```

✓ 8.9s Python

```
Your model makes 56 errors out of 1000
```

3. Make predictions on test data

Now let's see how many errors your model makes on the test set.

```python
# Compute log Pr(label|image) for each [test image,label] pair.
k = 10
score = np.zeros((len(test_labels),k))
for label in range(0,k):
    rv = multivariate_normal(mean=mu[label], cov=reg_sigma[label])
    for i in range(0,len(test_labels)):
        score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])
predictions = np.argmax(score, axis=1)
# Finally, tally up score
errors = np.sum(predictions != test_labels)
print("Your model makes " + str(errors) + " errors out of 10000")
print("Error rate: " +str(errors / len(test_labels)))
```

✓ 38.0s Python

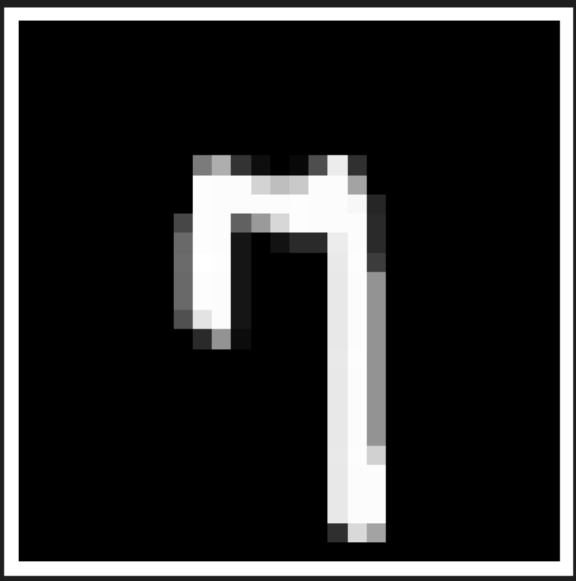
```
Your model makes 525 errors out of 10000
Error rate: 0.0525
```

```python
# display 5 misclassified test digits
num_display = 0
for i in range(len(predictions)):
    if predictions[i] != test_labels[i]:
        print('For the following image, predicted label was ' + str(predictions[i]) + ' but true label was ' + str(test_labels[i]))
        displaychar(test_data[i])
        num_display += 1
    if (num_display == 5):
        break
```

✓ 0.4s Python

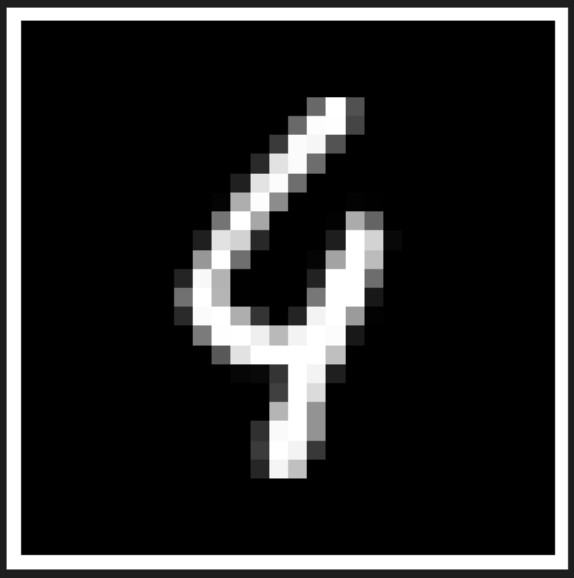
For the following image, predicted label was 7 but true label was 9

For the following image, predicted label was 9 but true label was 7

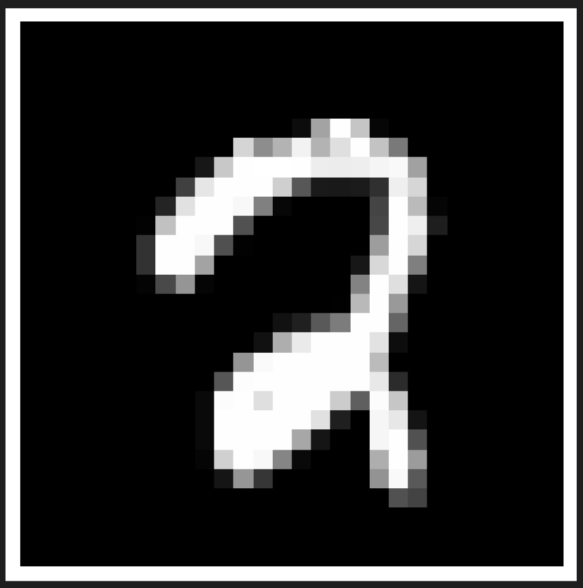


For the following image, predicted label was 1 but true label was 7

For the following image, predicted label was 9 but true label was 4



```
For the following image, predicted label was 9 but true label was 2
```



4. Quick exercises

You will need to answer variants of these questions as part of this week's assignment.

Exercise 1: What happens if you do not regularize the covariance matrices?
The code generates an error because the covariance matrix is singular

Exercise 2: What happens if you set the value of c too high, for instance to one billion? Do you understand why this happens?
If the 'c' value is way too high than the error rate also becomes very high. The reason for this is because multivariate gaussian density equation make use of the inverse of the regularized covariance matrix. If the diagonal of regularied covarince matrix is very large than its inverse will basicly become a matrix of zeros. This will then make the density function output a similar answer no matter the input which leads to misclassifying many inputs

Exercise 3: What value of c did you end up using? How many errors did your model make on the training set?
I ended up using a c value 10,000. The model made 525 errors out of the 10,000 test points which is an error rate of 5.25%.

If you have the time: We have talked about using the same regularization constant c for all ten classes. What about using a different value of c for each class? How would you go about choosing these? Can you get better performance in this way?