generative_mnist

January 26, 2025

1 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.

1.1 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.

[39]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import gzip, os, sys
import numpy as np
from scipy.stats import multivariate_normal

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

[40]:
```
# 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
```

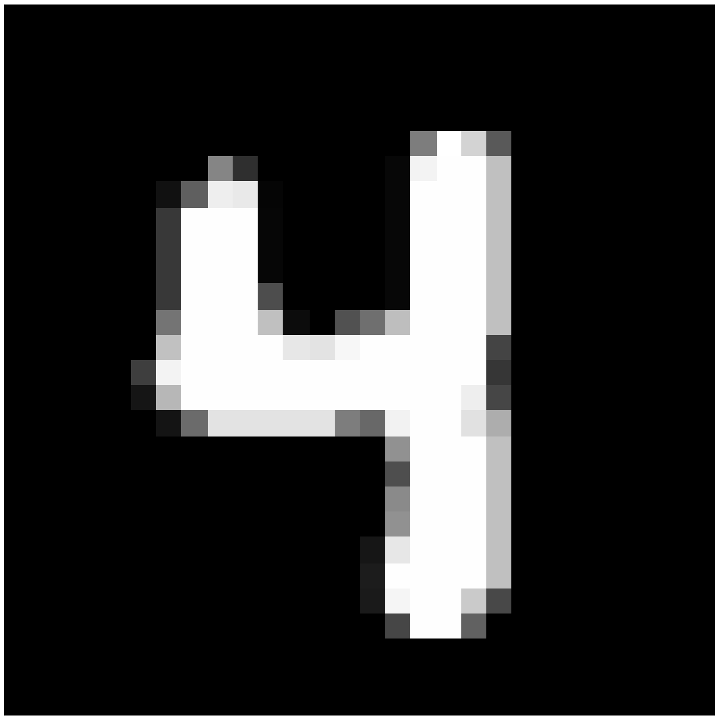
Now load in the training set and test set

[41]:
```
## 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')
```

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

[51]:
```
def displaychar(image):
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.axis('off')
    plt.show()
```

[52]:
```
displaychar(train_data[58])
```



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.

[44]:
```
train_data.shape, train_labels.shape
```

[44]:
```
((60000, 784), (60000,))
```

1.2 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.

[45]:
```
from sklearn.model_selection import train_test_split
```

[87]:
```
def compute_log_likelihood(X, mu, sigma, c):
    log_probs = []
    smoothed_sigma = {j: sigma[j] + c * np.eye(sigma[j].shape[0]) for j in␣
 ↪range(10)}

    for j in range(10):
        try:
            # Compute log-probability with the smoothed covariance matrix
            dist = multivariate_normal(mean=mu[j], cov=smoothed_sigma[j],␣
 ↪allow_singular=False)
            log_probs.append(dist.logpdf(X))
        except np.linalg.LinAlgError:
            print(f"Singular matrix encountered for class {j}, adjusting␣
 ↪regularization.")
```

```
            # If a singular matrix is encountered, apply more regularization
            smoothed_sigma[j] = sigma[j] + 1e-3 * np.eye(sigma[j].shape[0])  #␣
↪Increase regularization
            dist = multivariate_normal(mean=mu[j], cov=smoothed_sigma[j],␣
↪allow_singular=False)
            log_probs.append(dist.logpdf(X))

    return np.array(log_probs).T
```

[88]:
```
# def compute_log_likelihood_no_smoothing(X, mu, sigma, c):
#     log_probs = []
#     #new_sigma = []

#     for j in range(10):
#         try:
#             # Compute log-probability with the smoothed covariance matrix
#             dist = multivariate_normal(mean=mu[j], cov=sigma[j],␣
↪allow_singular=False)
#             log_probs.append(dist.logpdf(X))
#         except np.linalg.LinAlgError:
#             print(f"Singular matrix encountered for class {j}, adjusting␣
↪regularization.")
#             # If a singular matrix is encountered, apply more regularization
#             sigma[j] = sigma[j] + 1e-3 * np.eye(sigma[j].shape[0])  #␣
↪Increase regularization
#             dist = multivariate_normal(mean=mu[j], cov=new_sigma[j],␣
↪allow_singular=False)
#             log_probs.append(dist.logpdf(X))

#     return np.array(log_probs).T
```

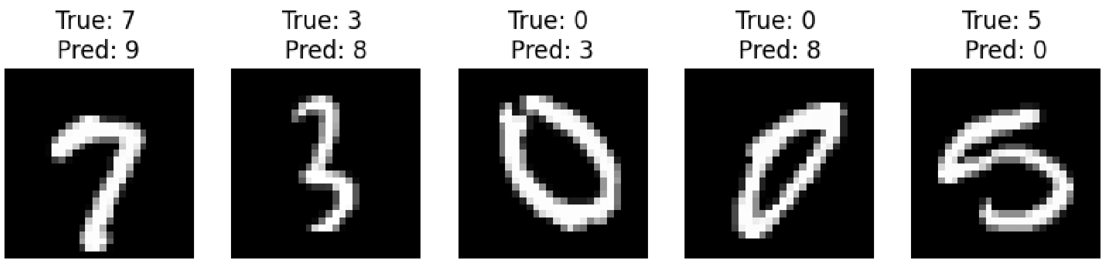
[89]:
```
def fit_generative_model(x,y):
    k = 10  # labels 0,1,...,k-1
    d = (x.shape)[1]  # number of features
    mu = np.zeros((k,d))
    sigma = np.zeros((k,d,d))
    pi = np.zeros(k)
    ###
    ### Your code goes here
    # split the training set of size 60000 into training set of 50000 and␣
↪validation set of 10000
    X_train, X_val, y_train, y_val = train_test_split(x, y, test_size=10000,␣
↪random_state=7)

    # class probability computations
    class_counts = np.bincount(y_train)
```

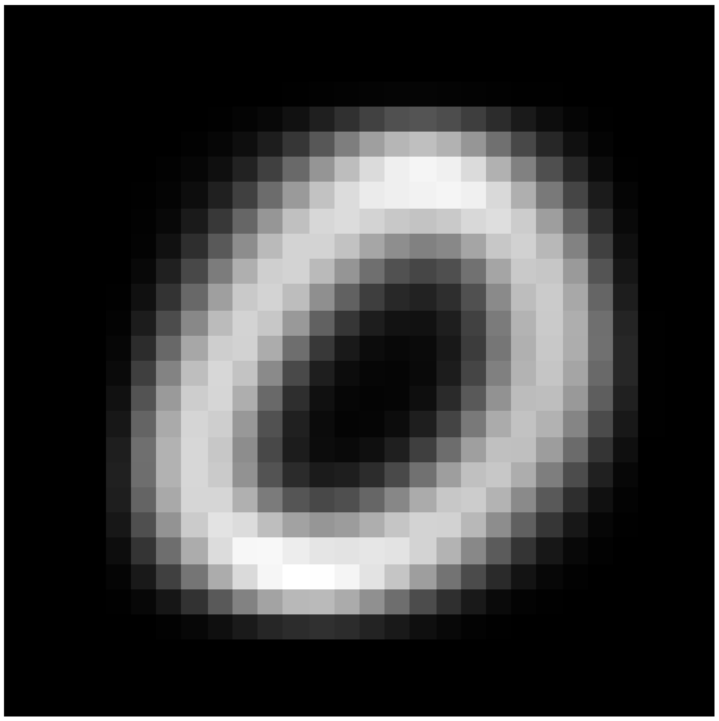
```python
    pi = class_counts / len(y_train)
    # print(X_train.shape)
    # print(class_counts)
    # print(pi)

    # compute mean and covariance matrix for each class
    mu = {}
    sigma = {}

    for j in range(k):
        class_data = X_train[y_train == j]
        mu[j] = np.mean(class_data, axis=0)
        sigma[j] = np.cov(class_data, rowvar=False)

    # grid search for best c
    candidate_c = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1]
    best_c = None
    lowest_val_error = float('inf')

    for c in candidate_c:
        log_probs = compute_log_likelihood(X_val, mu, sigma, c)
        predictions = np.argmax(np.log(pi) + log_probs, axis=1)
        val_error = np.mean(predictions != y_val)
        if val_error < lowest_val_error:
            lowest_val_error = val_error
            best_c = c

    print(f"Best c: {best_c}, Validation Error Rate : {lowest_val_error}")

    log_probs_test = compute_log_likelihood(test_data, mu, sigma, best_c)
    test_predictions = np.argmax(np.log(pi) + log_probs_test, axis=1)
    test_error = np.mean(test_predictions != test_labels)

    print(f"Test Error Rate: {test_error}")

    misclassified_indices = np.where(test_predictions != test_labels)[0]
    random_indices = np.random.choice(misclassified_indices, 5, replace=False)

    plt.figure(figsize=(10, 2))
    for i, idx in enumerate(random_indices):
        plt.subplot(1, 5, i + 1)
        plt.imshow(test_data[idx].reshape(28, 28), cmap='gray')
        plt.title(f"True: {test_labels[idx]} \nPred: {test_predictions[idx]}")
        plt.axis("off")
    plt.show()

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

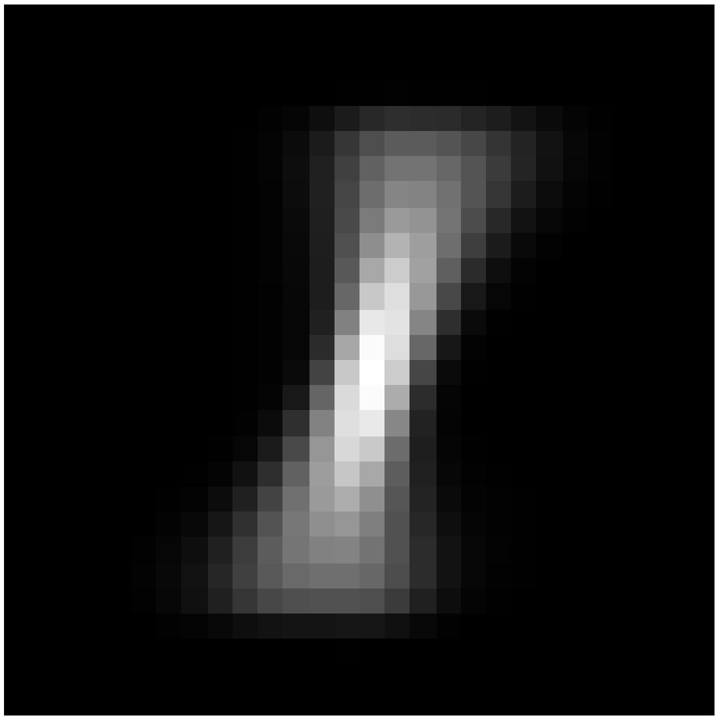
Okay, let's try out your function. In particular, we will use **displaychar** to visualize the means of the Gaussians for the first three digits. You can try the other digits on your own.

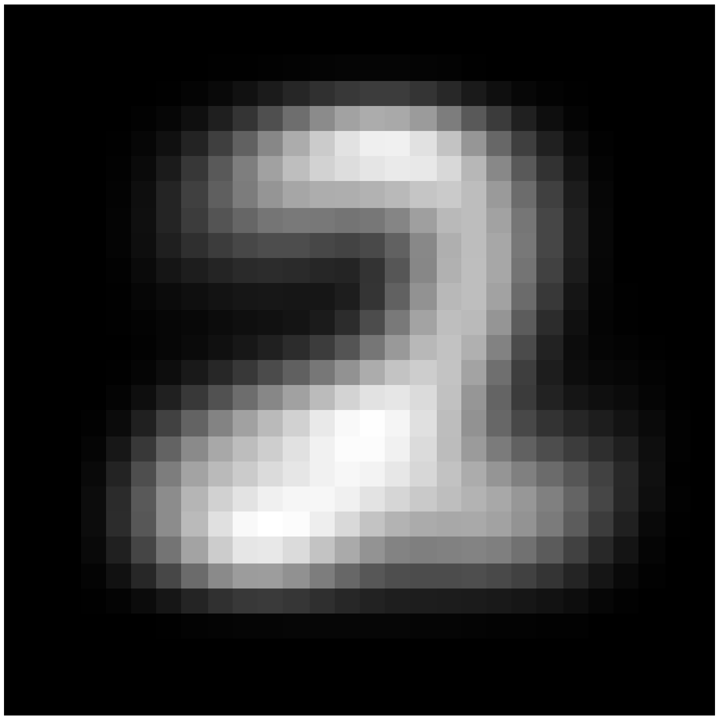
[90]:
```
mu, sigma, pi = fit_generative_model(train_data, train_labels)
displaychar(mu[0])
displaychar(mu[1])
displaychar(mu[2])
```

```
Singular matrix encountered for class 0, adjusting regularization.
Singular matrix encountered for class 1, adjusting regularization.
Singular matrix encountered for class 2, adjusting regularization.
Singular matrix encountered for class 3, adjusting regularization.
Singular matrix encountered for class 4, adjusting regularization.
Singular matrix encountered for class 5, adjusting regularization.
Singular matrix encountered for class 6, adjusting regularization.
Singular matrix encountered for class 7, adjusting regularization.
Singular matrix encountered for class 8, adjusting regularization.
Singular matrix encountered for class 9, adjusting regularization.
Singular matrix encountered for class 0, adjusting regularization.
Singular matrix encountered for class 1, adjusting regularization.
Singular matrix encountered for class 5, adjusting regularization.
Singular matrix encountered for class 6, adjusting regularization.
Best c: 1, Validation Error Rate : 0.1658
Test Error Rate: 0.1567
```









1.3 3. Make predictions on test data

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

[91]:
```
# 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=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")
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
Cell In[91], line 5
      3 score = np.zeros((len(test_labels),k))
      4 for label in range(0,k):
----> 5     rv = multivariate_normal(mean=mu[label], cov=sigma[label])
      6     for i in range(0,len(test_labels)):
```

```
      7         score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/scipy/stats/_multivariate.py:401, in multivariate_normal_gen.
 ↪__call__(self, mean, cov, allow_singular, seed)
    396 def __call__(self, mean=None, cov=1, allow_singular=False, seed=None):
    397     """Create a frozen multivariate normal distribution.
    398 
    399     See `multivariate_normal_frozen` for more information.
    400     """
--> 401     return multivariate_normal_frozen(mean, cov,
    402                                       allow_singular=allow_singular,
    403                                       seed=seed)

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/scipy/stats/_multivariate.py:908, in multivariate_normal_frozen
 ↪__init__(self, mean, cov, allow_singular, seed, maxpts, abseps, releps)
    865 """Create a frozen multivariate normal distribution.
    866 
    867 Parameters
   (…)
    904 
    905 """ # numpy/numpydoc#87  # noqa: E501
    906 self._dist = multivariate_normal_gen(seed)
    907 self.dim, self.mean, self.cov_object = (
--> 908     self._dist._process_parameters(mean, cov, allow_singular))
    909 self.allow_singular = allow_singular or self.cov_object._allow_singular
    910 if not maxpts:

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/scipy/stats/_multivariate.py:425, in multivariate_normal_gen.
 ↪_process_parameters(self, mean, cov, allow_singular)
    418 dim, mean, cov = self._process_parameters_psd(None, mean, cov)
    419 # After input validation, some methods then processed the arrays
    420 # with a `_PSD` object and used that to perform computation.
    421 # To avoid branching statements in each method depending on whether
    422 # `cov` is an array or `Covariance` object, we always process the
    423 # array with `_PSD`, and then use wrapper that satisfies the
    424 # `Covariance` interface, `CovViaPSD`.
--> 425 psd = _PSD(cov, allow_singular=allow_singular)
    426 cov_object = _covariance.CovViaPSD(psd)
    427 return dim, mean, cov_object

File /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
 ↪site-packages/scipy/stats/_multivariate.py:178, in _PSD.__init__(self, M,␣
 ↪cond, rcond, lower, check_finite, allow_singular)
    175 if len(d) < len(s) and not allow_singular:
    176     msg = ("When `allow_singular is False`, the input matrix must be "
    177            "symmetric positive definite.")
```

```
--> 178     raise np.linalg.LinAlgError(msg)
    179 s_pinv = _pinv_1d(s, eps)
    180 U = np.multiply(u, np.sqrt(s_pinv))

LinAlgError: When `allow_singular is False`, the input matrix must be symmetric
  ↪positive definite.
```

2 Part a

2.1 Split dataset into training and validation sets

randomly_shuffle_data()

train_set, validation_set = split_data(dataset, train_size=50000, val_size=10000)

2.2 Compute class probabilities (j) on the training set

class_counts = count_labels(train_set)

pi = {j: class_counts[j] / len(train_set) for j in range(10)}

2.3 Fit a Gaussian to each class

for j in range(10):

```
class_data = get_data_for_class(train_set, j)

mu[j] = compute_mean(class_data)

sigma[j] = compute_covariance(class_data)
```

2.4 Add smoothing to the covariance matrix

best_c = None

lowest_validation_error = float("inf")

for c in candidate_values_of_c:

```
smoothed_sigma = {j: sigma[j] + c * identity_matrix(784) for j in range(10)}

# Evaluate on the validation set
validation_error = compute_validation_error(validation_set, pi, mu, smoothed_sigma)

if validation_error < lowest_validation_error:
    lowest_validation_error = validation_error
    best_c = c
```

2.5 Use the chosen value of c to finalize the model

final_smoothed_sigma = {j: sigma[j] + best_c * identity_matrix(784) for j in range(10)}

3 Classify test set digits

test_error, misclassified_indices = evaluate_on_test_set(test_set, pi, mu, final_smoothed_sigma)

4 Display five randomly selected misclassified digits

randomly_sample_and_display(test_set, misclassified_indices, n=5)

5 Part b

I used a single value of c for all ten classes. The value of c I got after testing these values: [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1] was 1.

6 Part c

The test error rate when I used c = 1 was 0.1567.

7 Part d

This was done above.

7.1 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?

Exercise 2: What happens if you set the value of `c` too high, for instance to one billion? Do you understand why this happens?

Exercise 3: What value of c did you end up using? How many errors did your model make on the training set?

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?

8 Exercise 1

If you don't regularize the covariance matrices, you get an error because the input matrix is not symmetric positive definite, which means it is a singular matrix.

9 Exercise 2

When the value of c was set too high, like 1000000000, the error on the test set was extremely high. I'm not one hundred percent sure, but I believe that this happens because it can bias the model towards the independence assumption and can lead to oversmoothing.

10 Exercise 3

I used c=1, and it had an error rate of .1567 on the test set.

11 Exercise 4

I don't have the time now, but I will try this in the future.