

The results for the most part look like plausible digits from the dataset!

Consider what we've done here: given a sampling of handwritten digits, we have modeled the distribution of that data in such a way that we can generate brand new samples of digits from the data: these are “handwritten digits” that do not individually appear in the original dataset, but rather capture the general features of the input data as modeled by the mixture model. Such a generative model of digits can prove very useful as a component of a Bayesian generative classifier, as we shall see in the next section.

## In-Depth: Kernel Density Estimation

In the previous section we covered Gaussian mixture models (GMM), which are a kind of hybrid between a clustering estimator and a density estimator. Recall that a density estimator is an algorithm that takes a  $D$ -dimensional dataset and produces an estimate of the  $D$ -dimensional probability distribution which that data is drawn from. The GMM algorithm accomplishes this by representing the density as a weighted sum of Gaussian distributions. *Kernel density estimation* (KDE) is in some senses an algorithm that takes the mixture-of-Gaussians idea to its logical extreme: it uses a mixture consisting of one Gaussian component *per point*, resulting in an essentially nonparametric estimator of density. In this section, we will explore the motivation and uses of KDE. We begin with the standard imports:

```
In[1]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

### Motivating KDE: Histograms

As already discussed, a density estimator is an algorithm that seeks to model the probability distribution that generated a dataset. For one-dimensional data, you are probably already familiar with one simple density estimator: the histogram. A histogram divides the data into discrete bins, counts the number of points that fall in each bin, and then visualizes the results in an intuitive manner.

For example, let's create some data that is drawn from two normal distributions:

```
In[2]: def make_data(N, f=0.3, rseed=1):
        rand = np.random.RandomState(rseed)
        x = rand.randn(N)
        x[int(f * N):] += 5
        return x

x = make_data(1000)
```