Assignment 2

Please fill out the relevant cells below according to the instructions. When done, save the notebook and export it to PDF, upload both the `ipynb` and the PDF file to Canvas.

Group Members

Group submission is highly encouraged. If you submit as part of group, list all group members here. Groups can comprise up to 4 students.

- Vanessa Roser
- Declan Campbell

In [2]:
```
from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
```

Problem 1: Design the Optimal Peak Detection System (3pts)

A common problem in the analysis event data is the precise localization of a peak with a known form, say a Gaussian. We generally have two options.

1. We can perform direct density estimation given the known parametric form, which is equivalent to a GMM with $K = 1$.
2. We can form a histogram, which turns event locations into counts per bin, and then fit a Gaussian to the pairs (x_k, N_k) of (mean) bin location and counts.

Side note: For image analysis, where the peak could be e.g. a small tree on a satellite photo, we have don't have that choice. It's always option 2 because the incoming photons are automatically binned into pixels by the detector in the camera.

Step 1 (2pts)

Compare the uncertainty of the center estimation in these two cases as a function of the total event number N. Specifically, assume a standard normal distribution (i.e. $\sigma = 1$) for the generating process. Draw N events.

For case 1, determine the mean $\tilde{\mu}$ of the event distribution. For case 2, bin the samples with a bin width $\Delta = 1$. That gives you a set of bin centers and counts $\mathcal{D} = \{(x_1, N_1), \ldots, (x_K, N_K)\}$. The likelihood for each bin is Poissonian, with a mean rate λ_k that follows a Gaussian parametric form. But the Gaussian shape also needs to be integrated in the bins:

$$\lambda_k = \frac{N}{\sqrt{2\pi}\sigma} \int_{x_k-\Delta/2}^{x_k+\Delta/2} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) dx = \frac{N}{2}\left[\mathrm{erf}\left(\frac{\Delta/2-(x_k-\mu)}{\sqrt{2}\sigma}\right) + \mathrm{erf}\left(\frac{x_k}{}\right.\right.$$

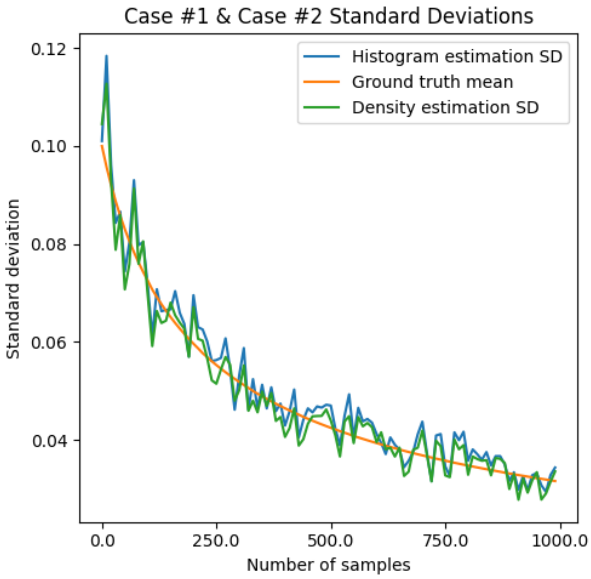
for some unknown μ. erf stands for the Error function. Use `scipy.optimize` to find the MLE $\tilde{\mu}_b$.

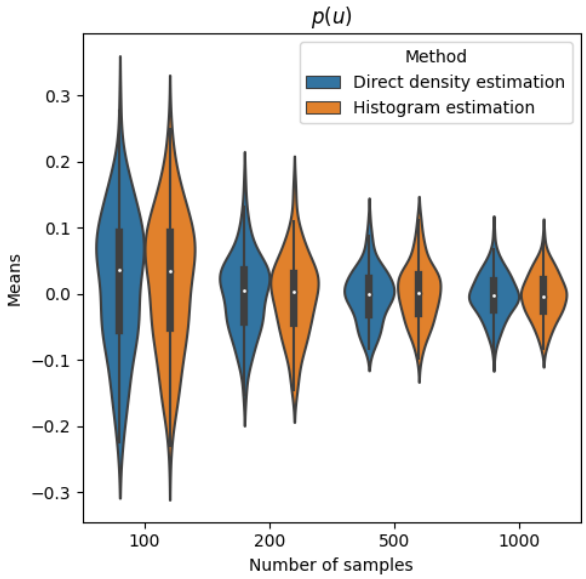
Repeat the process multiple times, plot $p(\tilde{\mu})$ and $p(\tilde{\mu}_b)$, and compute their mean and variances. Repeat with different N.

In [3]:
```
from scipy.special import erf
from scipy.optimize import minimize
import pandas as pd

sigma = 1

def get_lambda(mu, x, Ncounts, sigma):
    lam = np.sum(Ncounts)/2*(erf((0.5-(x - mu)) / (np.sqrt(2)*sigma)) + erf((x-mu+0
    return lam

def get_lkhd(mu, x, Ncounts, sigma, delta=1):
    lam = get_lambda(mu, x, Ncounts, sigma)
    lkhd = -sum(Ncounts*np.log(lam) - lam*delta)
    return lkhd

n_samples = 100
n_repetitions = 100
samples = np.linspace(100,1000,n_samples,dtype=int)
outputs = np.zeros([n_samples, n_repetitions])
means = np.zeros([n_samples, n_repetitions])
for i, n in enumerate(samples):
    for ir in range(n_repetitions):
        data = np.random.randn(n)
        Ncounts, x = np.histogram(data, bins=np.arange(-6,7))
        x = (x[:-1] + x[1:]) / 2
        minimum = minimize(get_lkhd, x0=0, args=(x, Ncounts, sigma))
        outputs[i, ir] = minimum.x[0]
        means[i, ir] = data.mean()
```

In [4]:
```
import seaborn as sns

# Plot the standard deviation.
fig, axes = plt.subplots(1, 2, figsize=(10, 5), tight_layout=True)
axes[0].plot(np.std(outputs, axis=1), label='Histogram estimation SD')
axes[0].plot(1/np.sqrt(samples), label='Ground truth mean')
axes[0].plot(np.std(means, axis=1), label='Density estimation SD')
axes[0].legend()
axes[0].set_title('Case #1 & Case #2 Standard Deviations')
axes[0].set_xlabel('Number of samples')
axes[0].set_xticks(np.linspace(0, 100, 5), np.linspace(0, 1000, 5))
axes[0].set_ylabel('Standard deviation')

# Plot violin plots of p(u) and p(u_b).
inds = [0, 11, 44, -1]
ns = np.concatenate([np.repeat(samples[ind], 100) for ind in inds])
ns = np.concatenate([ns, ns])
estimates = np.concatenate([means[inds, :].ravel(), outputs[inds, :].ravel()])
method = np.concatenate([np.repeat('Direct density estimation', 400), np.repeat('Hi
df = pd.DataFrame({'estimates': estimates, 'nsamples': ns, 'Method': method})
sns.violinplot(x='nsamples', y='estimates', hue='Method', data=df, ax=axes[1])
axes[1].set_xlabel('Number of samples')
axes[1].set_ylabel('Means')
axes[1].set_title(r'$p(u)$')
```

Out[4]:
```
Text(0.5, 1.0, '$p(u)$')
```





Problem 2: Clustering Hyper-Spectral Images (3pts)

Hyper-spectral images of a scene are recorded in hundreds of wavelengths, typically extending beyond the range perceptible by humans. They play a critical role in remote sensing from aerial and satellite platforms because they allow us to infer e.g. where roads or vegetation are (even under clouds), how well crops grow, the salinity of water...

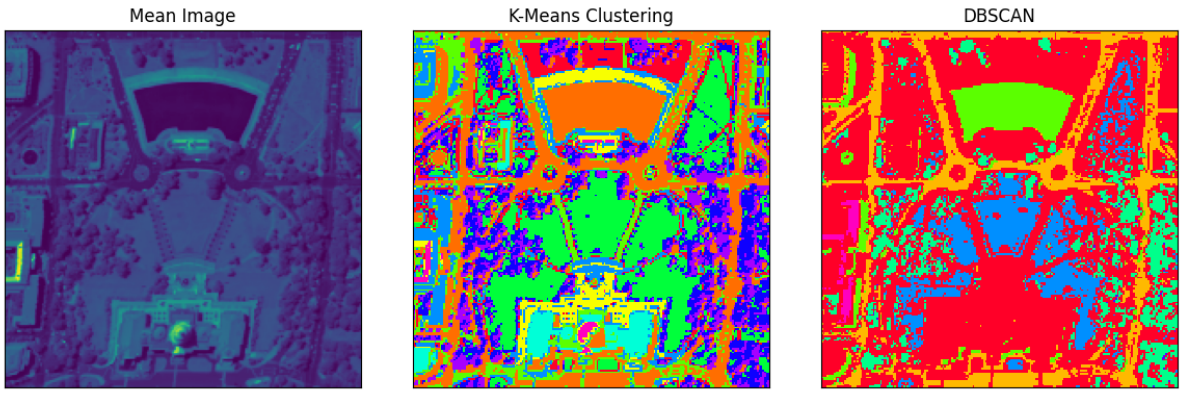
Often, we don't know a priori what is recorded in a particular hyper-spectral data set. Unsupervised clustering is then a way to identify interesting structures. Download this hyper-spectral data set, taken from an airplane flying over Capitol Hill in Washington, D.C. It consists of 191 spectral channels, each having 200 x 200 pixels. Then pick 2 different clustering algorithms and attempt to identify interesting structures. This will typically require some tinkering with parameter settings. When done, compare what the two clustering algorithms found and try to explain why the outcomes differ on the basis of the assumptions made by the algorithms.

Hint: The data are stored in the layout `(N_channels, N_pixels)`, with each 2D image flattened into a single long vector. This treats every channel as an independent sample with a vector of intensity variations per pixel. Alternatively, you can flip the axes into the layout `(N_pixels, N_channels)`, which treat every pixel as an independent sample with a vector of intensity variations per channel. Both of these are valid, as are hybrids and subsets. Decide which of them you want to use.

Hint 2: For visualization, it's best to reshape the pixel vector from `(N_pixels)` to `(N_pixels_vertical, N_pixels_horizontal)`. Also, Google Map/Earth could turn out to be useful for your visual orientation.

In [5]:
```
from sklearn.cluster import DBSCAN, KMeans

X = np.load('subset.npy').T
fig, axes = plt.subplots(1, 3, figsize=(12, 4), tight_layout=True)

# Ground truth image.
mean_img = X.reshape(200, 200, 191).mean(2)
axes[0].imshow(mean_img)
axes[0].set_title('Mean Image')
axes[0].set_xticks([])
axes[0].set_yticks([])

# K-means clustering.
km_labels = KMeans(n_clusters=10).fit_predict(X)
axes[1].imshow(km_labels.reshape(200,200), cmap='gist_rainbow', interpolation='none
axes[1].set_title('K-Means Clustering')
axes[1].set_xticks([])
axes[1].set_yticks([])

# DBSCAN clustering.
dbscan_labels = DBSCAN(eps=1000, min_samples=100).fit_predict(X)
axes[2].imshow(dbscan_labels.reshape(200,200), cmap='gist_rainbow', interpolation='
axes[2].set_title('DBSCAN')
axes[2].set_xticks([])
axes[2].set_yticks([])
```

Out[5]:
```
[]
```



K-means clustering is a hard clustering algorithm that assigns each data point to a single cluster based on its proximity to the cluster's centroid, and attempts to minimize the sum of squared distances between each data point and its assigned cluster centroid.. K-means assumes that the variance of each cluster is spherical and equivalent, which is likely why the image output contains roughly equal distributions of each cluster, rather than clusters of dramatically different size.

On the other hand, DBSCAN is a non-parametric clustering algorithm that clusters points within neighborhoods as determined by the distance set by the epsilon parameter, which specifies the radius of the neighborhood. DBSCAN can describe non-linearly seperable clusters, and is robust to noise and outliers. Here, epsilon was set by thresholding the kth-nearest neighbor plots for the dataset, and the minimum number of points to define a cluster was set as 100 to prevent arbitrarily small clusters from forming. DBSCAN seems to prioritize a solution with a smaller number of clusters that we arbitrarily set for Kmeans, but still reveals the general forms contained the image data.

Problem 3: Survey Responses - The Good, the Bad, and the Ugly (4pts)

Questionnaires, especially online, are often contaminated with incorrect answers, which come in two main forms: malice or lack of interest. For any single question, it is impossible to infer the motivations of a user from their response, but with multiple questions it should be doable. Let's find out with the data in `questionnaire.npy`, which contains 7 Yes/No questions for 100 users.

Step 1 (3pts)

Assume a mixture model with 3 groups:

- the Good G: users who try to answer the questions
- the Bad B: users who don't seem to pay attention
- the Ugly U: users who intentionally try to answer the questions incorrectly

Because of the binary nature of the question, the base distributions are of the Binomial type.

Review Murphy 11.2.2 and Exercise 11.3. Then code your own EM algorithm to solve for the posterior weights π_k and mean probabilities μ_{jk} of each question. Think about whether you can put non-trivial priors on the component weights.

Try to identify which of the components describes G. Then report the the posterior weights as well as the mean and variances of the Binomial distribution parameter μ_j for every question j *only* for the users you believe to be in G.

Hint: Not every user has the same behavior and motivation. Think about how each user group is expected to perform in the survey, and decide which one is G. Plotting the means might help.

Hint 2: Pay attention to the value of the μ_k. The log likelihood is ill-defined if $\mu_k \in \{0, 1\}$.

mu: for each state probability each person answers yes N: number of questions pi: base probabilities of each class

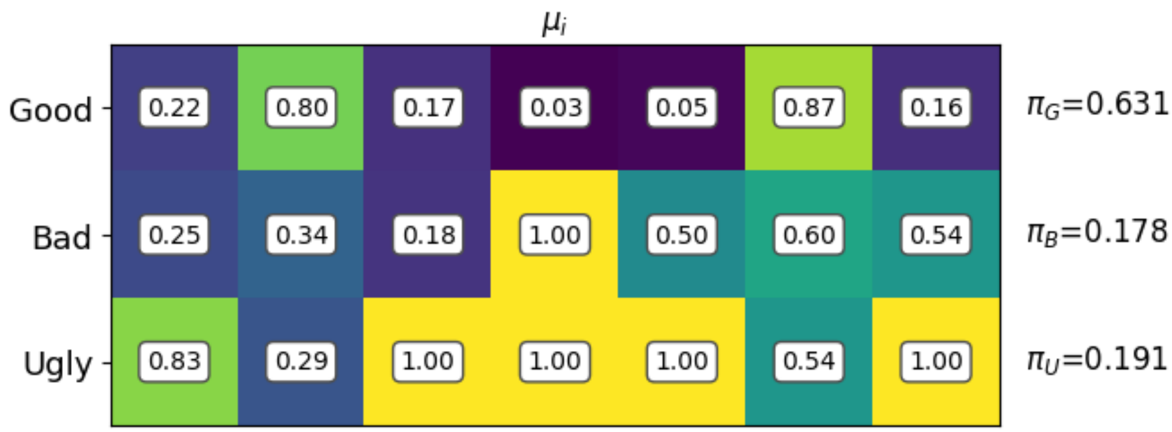
In [8]:

```python
def e_step(mu, pi, X):
    N, K = X.shape[0], pi.shape[0]
    z_i = np.zeros([N, K])
    for n in range(N):
        for k in range(K):
            z_i[n, k] = np.prod(mu[k]**X[n] * (1-mu[k])**(1-X[n])) * pi[k]
        z_i[n,:] = z_i[n,:] / z_i[n,:].sum()
    return z_i


def m_step(z, X):
    K, M = z.shape[1], X.shape[1]
    mu_i = np.zeros([K, M])
    for k in range(K):
        for m in range(M):
            mu_i[k, m] = np.sum(X[:,m] * z[:,k]) / z[:,k].sum()
    pi_i = z.sum(0) / z.shape[0]
    return mu_i, pi_i

# Run the analysis.
from tqdm import tqdm
X = np.load('questionnaire.npy')
pi_i = np.ones(3) / 3
mu_i = np.random.random(size=(3, 7))

n_steps = 1000
for i in tqdm(range(n_steps)):
    z_i = e_step(mu_i, pi_i, X)
    mu_i, pi_i = m_step(z_i, X)
```

```
100%|██████████████████████████████████████████████████████████████████████████████████████████████████| 1000/1000 [00:03<00:00, 250.35it/s]
```

In [9]:

```python
fig, ax = plt.subplots(1, 1)
ax.matshow(mu_i, cmap='viridis')
ax.set_yticks(np.arange(mu_i.shape[0]), ['Good', 'Bad', 'Ugly'], fontsize=13)
ax.set_xticks([])
ax.set_title(r'$\mu_i$')
ax.text(6.7, 0.05, f'$\pi_G$={pi_i[0]:0.3f}', fontsize=12)
ax.text(6.7, 1.05, f'$\pi_B$={pi_i[1]:0.3f}', fontsize=12)
ax.text(6.7, 2.05, f'$\pi_U$={pi_i[2]:0.3f}', fontsize=12)

for (i, j), z in np.ndenumerate(mu_i):
    ax.text(j, i, '{:0.2f}'.format(z), ha='center', va='center', bbox=dict(boxstyle
```



Step 2 (1pts)

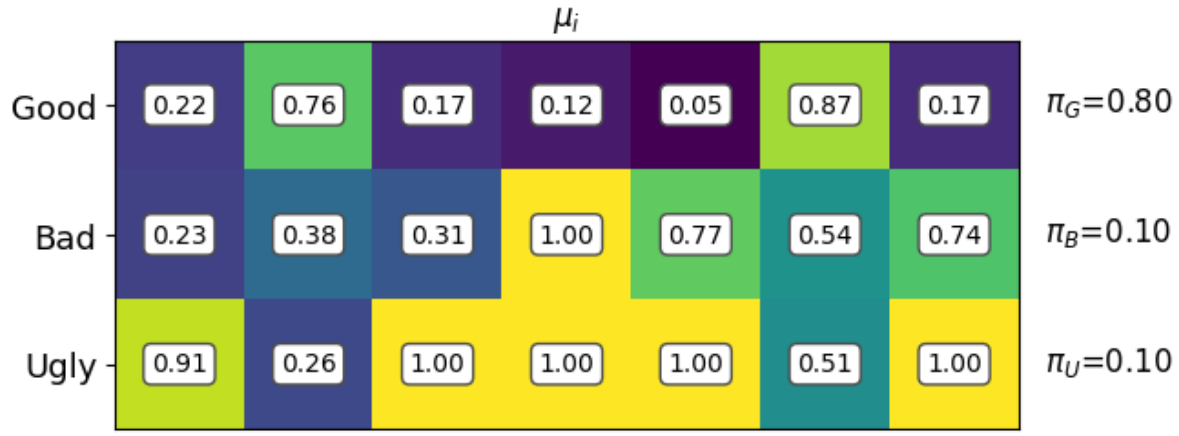
Now that you have identified the groups, rerun the the EM with a prior on the cluster assignments: $p(z_i = G) = 0.8$, $p(z_i = B) = 0.1$, $p(z_i = U) = 0.1 \forall i$. Plot the results.

Can you construct an alternative estimator of μ_j using not just the result for G? If so, what is the analytic form of this estimator?

In [12]:
```
X = np.load('questionnaire.npy')
pi = np.array([0.8, 0.1, 0.1])
mu_i = np.random.random(size=(3, 7))

n_steps = 1000
for i in tqdm(range(n_steps)):
    z_i = e_step(mu_i, pi, X)
    mu_i, _ = m_step(z_i, X)
```

```
100%|██████████████████████████████████████████████████| 1000/1000 [00:04<00:00, 246.56it/s]
```

In [13]:
```
fig, ax = plt.subplots(1, 1)
ax.matshow(mu_i, cmap='viridis')
ax.set_yticks(np.arange(mu_i.shape[0]), ['Good', 'Bad', 'Ugly'], fontsize=13)
ax.set_xticks([])
ax.set_title(r'$\mu_i$')
ax.text(6.7, 0.05, f'$\pi_G$={pi[0]:0.2f}', fontsize=12)
ax.text(6.7, 1.05, f'$\pi_B$={pi[1]:0.2f}', fontsize=12)
ax.text(6.7, 2.05, f'$\pi_U$={pi[2]:0.2f}', fontsize=12)

for (i, j), z in np.ndenumerate(mu_i):
    ax.text(j, i, '{:0.2f}'.format(z), ha='center', va='center', bbox=dict(boxstyle
```



We can construct an estimator for G using information from the bad actors (UGLY), inferring the μs using the opposing responses from the ugly group:

$$\mu_j = \frac{\mu_j^G * p(z_i{=}G) + (1-\mu_j^U) * p(z_i{=}U)}{p(z_i{=}G) + p(z_i{=}U)}$$