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 5 students.

- Aizhan Akhmetzhanova
- Tony Chen
- Lucas Makinen
- Emre Oezkan
- Sachin Smart

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. We can either perform direct density estimation given the known parametric form, which is equivalent to a GMM with $K = 1$. Or 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. For image analysis, where the peak could be e.g. a blip on a radar monitor or an unresolved person on a satellite photo, we have don't have that choice because the incoming photons are automatically binned into pixels by our cameras.

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. Determine the center $\tilde{\mu}$ of the event distribution.

Then, 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:

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

for some unknown μ. erf stands for the [Error function (https://en.wikipedia.org/wiki/Error_function)](https://en.wikipedia.org/wiki/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 variances. Repeat with different N.

Begin with the construction of a direct estimator $\hat{\mu}$ of the center μ of the samples. Let $(X_i)_{i\in\{1,\dots,N\}}$ be i.i.d. random variables with

$$X_i \sim \mathcal{N}(\mu, \sigma^2)$$

with parameters μ and σ. We assume a known variance, so that we only have to estimate μ. The MLE estimator $\hat{\mu}$ of μ is given by

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} X_i$$

We, therefore, just have to calculate the sample average.

Next, we consider a grid $(x_i)_{i\in\{1,\dots,K\}}$ with $x_1 \leq \dots \leq x_K$, and for $k \in \{1, \dots, K\}$, set

$$N_k = \sum_{i=1}^{N} \mathbb{1}_{\left\{x_k - 0.5\Delta \leq X_i < x_k + 0.5\Delta\right\}}$$

Then, we know that

$$N_k \sim \text{Poisson}(\lambda_k)$$

with

$$\lambda_k = \frac{N}{2}\left[\text{erf}\left(\frac{\Delta/2 - (x_k - \mu)}{\sqrt{2}\sigma}\right) + \text{erf}\left(\frac{x_k - \mu + \Delta/2}{\sqrt{2}\sigma}\right)\right]$$

Therefore, the maximum likelihood function, $L_k(\mu)$, corresponding to the k^{th} bin is given by

$$\log L_k(\mu) = N_k \log(\lambda_k) - \lambda_k$$

We obtain the maximum likelihood estimator $\hat{\mu}_k$ corresponding to the k^{th} bin by

$$\hat{\mu}_k = \arg\max_{\mu} L_k(\mu)$$

We define the final estimator $\hat{\mu}_b$ of μ as

$$\hat{\mu}_b = \frac{1}{K}\sum_{k=1}^{K} \hat{\mu}_k$$

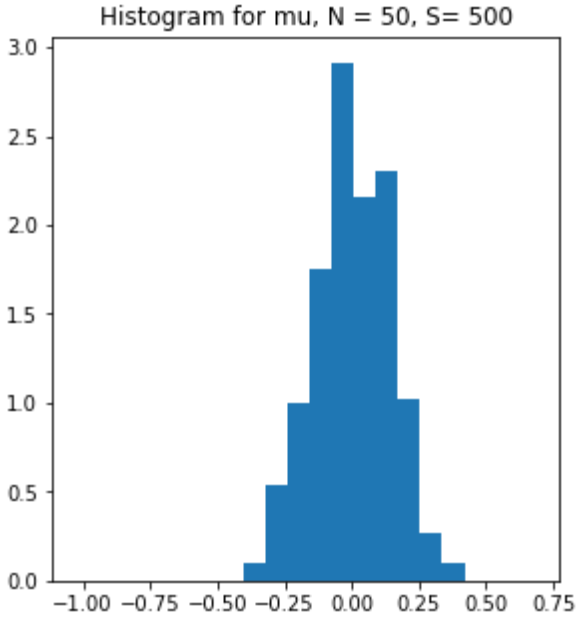
We implement the above for $S \in \mathbb{N}$ simulations to obtain a sample distribution of $\hat{\mu}$ and $\hat{\mu}_b$.

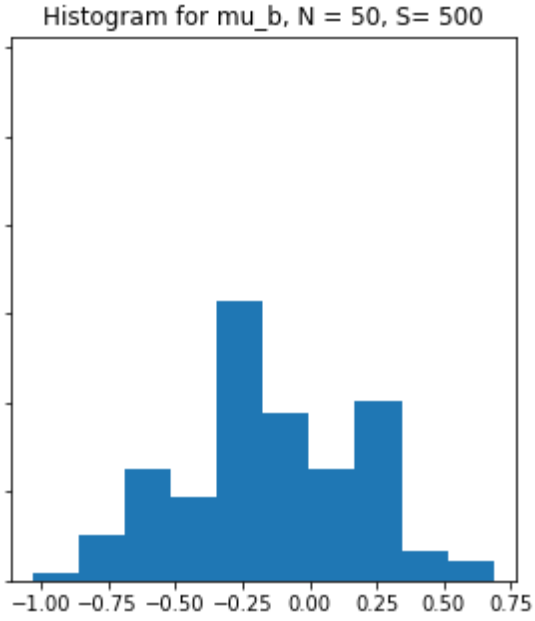
In [1]:
```
import numpy as np
from scipy.special import erf
from scipy.stats import norm
from scipy.stats import poisson
from scipy.optimize import minimize
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KernelDensity
import matplotlib.pyplot as plt
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')
```

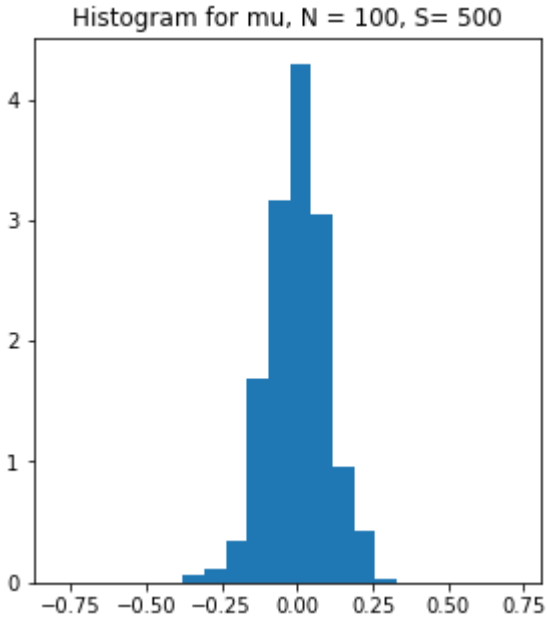
In [2]:
```
def sim(N = 100, S = 500, dt = 1.0):

    X = np.random.normal(loc = 0, scale = 1, size = N*S).reshape((N,S))

    # Sample mean
    mu = np.mean(X,axis = 0)

    # Grid
    x = np.arange(np.min(X), np.max(X)+dt, dt)
    N_v = np.zeros((len(x), S))

    for s in range(S):
        N_v[:,s] = np.vectorize(lambda x: np.sum(np.logical_and(X[:,s] >
= x-0.5*dt,X[:,s] <x+0.5*dt)))(x)

    def loglikelihood(x_k, N_k, mu):
        lmbd_k = N/2*(erf((0.5*dt-x_k+mu)/np.sqrt(2))+erf((x_k-mu+0.5*dt
)/np.sqrt(2)))
        return( N_k *np.log(lmbd_k) - lmbd_k )

    mu_b = np.zeros((len(x), S))
    for k in range(len(x)):
        for s in range(S):
            mu_b[k,s] = minimize(lambda mu: -loglikelihood(x[k],N_v[k,s
],mu), 0.0).x[0]
    mu_b = np.mean(mu_b, axis = 0)

    return mu, mu_b
```

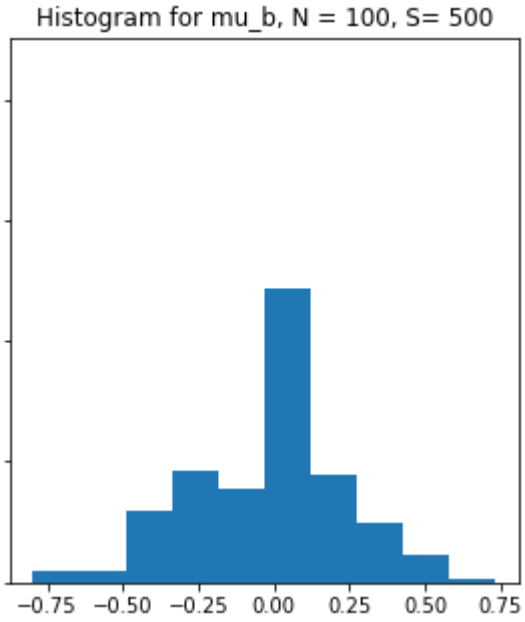
In [3]:

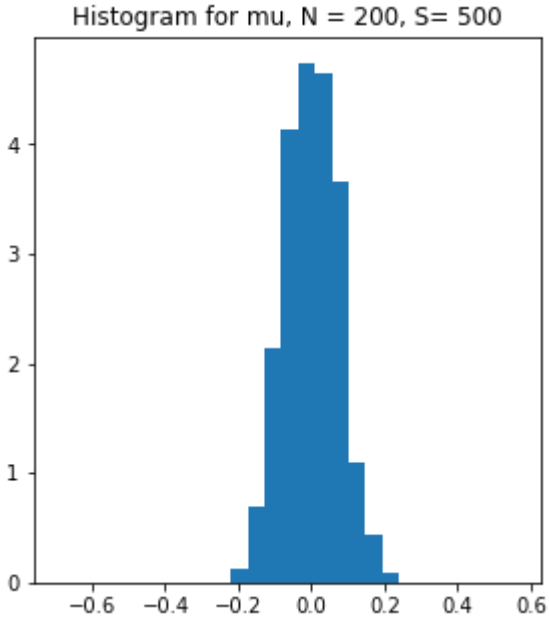
```
# Plotting
Ns = [50,100,200,400,800]
stats = np.zeros((len(Ns), 4))

for i,N in enumerate(Ns):
    mu, mu_b = sim(N = N)
    fig, axes = plt.subplots(1, 2, figsize=(10, 5), sharex=True, sharey=
True)
    axes[0].hist(mu, density = True)
    axes[0].title.set_text("Histogram for mu, N = "+str(N)+", S= "+str(5
00))
    axes[1].hist(mu_b, density = True)
    axes[1].title.set_text("Histogram for mu_b, N = "+str(N)+", S= "+str
(500))
    plt.show()
    stats[i, 0] = np.mean(mu)
    stats[i, 1] = np.mean(mu_b)
    stats[i, 2] = np.var(mu)
    stats[i, 3] = np.var(mu_b)

plt.plot(Ns, stats[:,2])
plt.plot(Ns, stats[:,3])
plt.title("Sample variance of estimators vs. N, S= "+str(500))
plt.ylabel("Sample Variance")
plt.xlabel("Number of events N")
plt.legend(("mu", "mu_b"))
plt.show()
```

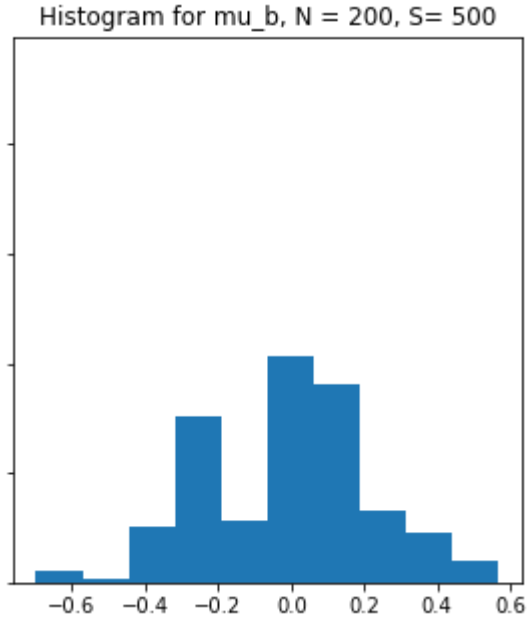

Histogram for mu, N = 50, S= 500
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75

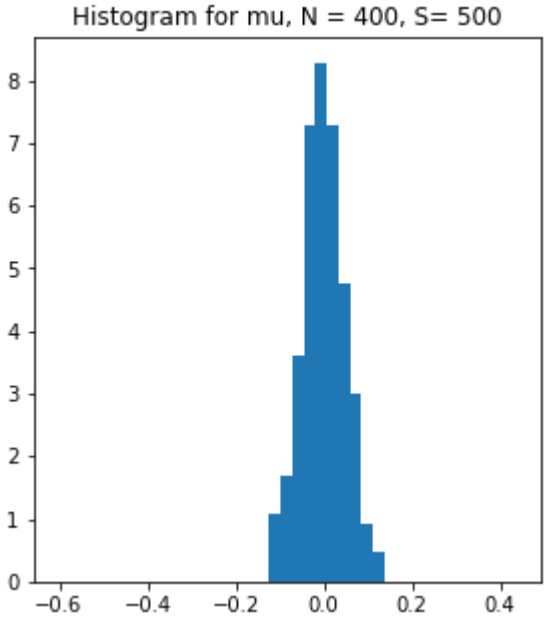

Histogram for mu_b, N = 50, S= 500
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75

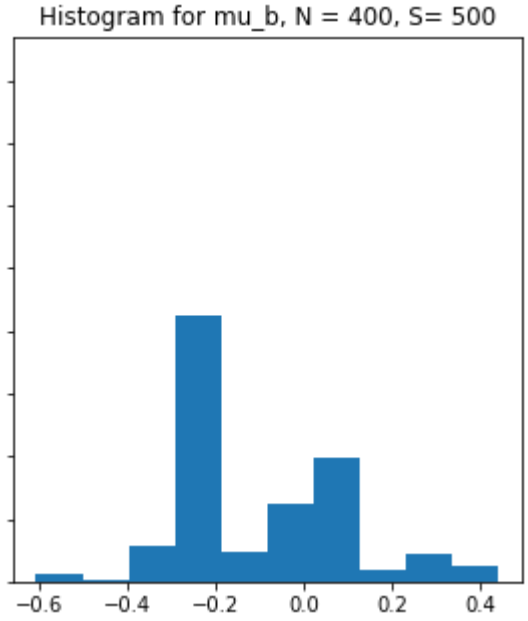

Histogram for mu, N = 100, S= 500
4
3
2
1
0
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75

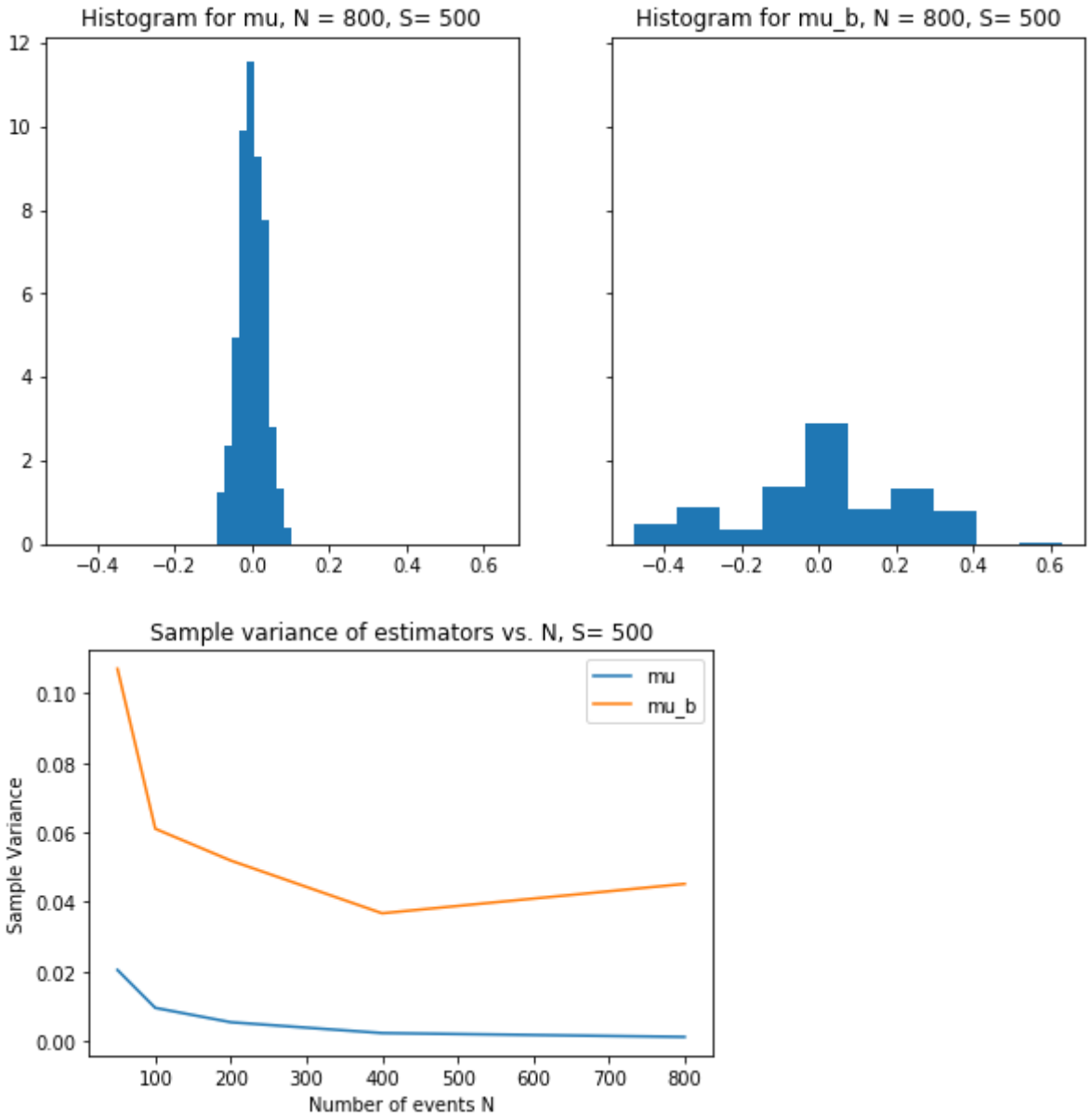

Histogram for mu_b, N = 100, S= 500
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75


Histogram for mu, N = 200, S= 500
0
1
2
3
4
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6


Histogram for mu_b, N = 200, S= 500
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6


Histogram for mu, N = 400, S= 500
0
1
2
3
4
5
6
7
8
−0.6
−0.4
−0.2
0.0
0.2
0.4


Histogram for mu_b, N = 400, S= 500
−0.6
−0.4
−0.2
0.0
0.2
0.4



We observe the following:

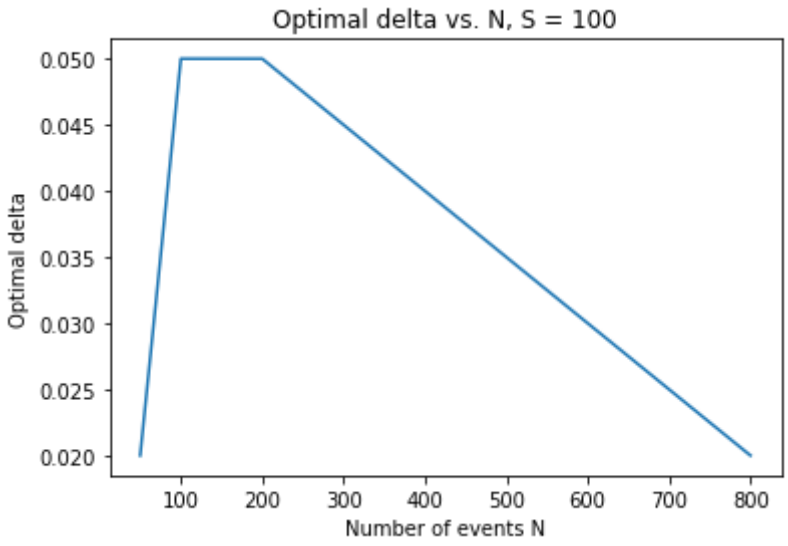
- The sample distribution of $\hat{\mu}$ folllows a normal distribution, which is something we mathematically expect.
- The sample distribution of $\hat{\mu}_b$ looks somewhat normal. However, there are some spikes and non-normalities observable.
- The sample variance of $\hat{\mu}_b$ is larger than the sample variance of $\hat{\mu}$. This is something we expect given that by binning data, we give up some information.

Step 2 (1pt)

Assume that your instrument only reports binned event counts. Determine the optimal bin width Δ as a function of the sample size N.

We follow the approach above for different choices of Δ and N. For each N, we determine the optimal Δ and plot the result. In this context, the optimal Δ is given by the bandwidth that minimizes the sample variance of $\hat{\mu}_b$.

In [ ]:
```
dts = np.arange(0.01, 0.1, 0.01)
Ns = [50,100,200,400,800]
S = 100
sts = np.zeros((len(dts), len(Ns)))

for i,N in enumerate(Ns):
    sts[:,i] = np.vectorize(lambda dt: np.var(sim(N=N,S=S,dt=dt)[1]))(dt
s)

dt_hat = dts[np.apply_along_axis(np.argmin, axis = 0, arr = sts)]

plt.plot(Ns, dt_hat)
plt.title("Optimal delta vs. N, S = "+str(S))
plt.xlabel("Number of events N")
plt.ylabel("Optimal delta")
```

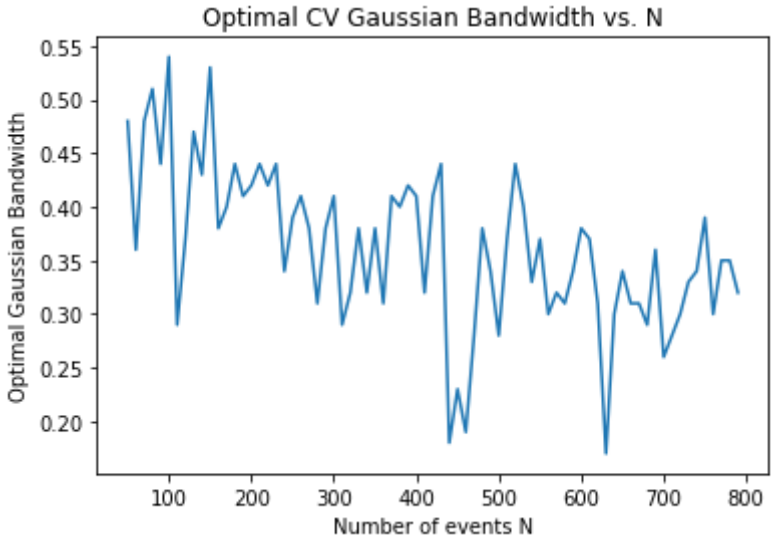


We make the following observations:

- For sufficiently larger N, optimal Δ decreases as a function of N. This makes sense. For larger sample sizes, we can choose smaller bin sizes while maintaining a relatively smooth histogram. A smooth histogram translates into a relatively low sample variance for $\hat{\mu}_b$.
- The curve for small sample sizes is too noisy to be interpretable. Therefore, we do not assign any qualitative interpretation into the initial increase of the curve.

As a comparison, we can use cross-validation to determine an optimal Gaussian bandwith for each different sample size NN. The optimal Gaussian Bandwidth is comparable to the optimal binsize in a histogram. We therefore expect to obtain a similar graph for the optimal Gaussian bandwith. We are implementing this below.

In [ ]:
```
params = {'bandwidth': np.arange(0.01, 1.01, 0.01)}
Ns = np.arange(50,800,10)
K = 1000
dts = np.zeros(len(Ns))

for i,N in enumerate(Ns):
    X = np.random.normal(0,1,N).reshape((-1,1))
    grid = GridSearchCV(KernelDensity(kernel = "gaussian"), params)
    grid.fit(X)
    dts[i] = grid.best_params_['bandwidth']

plt.plot(Ns,dts)
plt.title("Optimal CV Gaussian Bandwidth vs. N")
plt.xlabel("Number of events N")
plt.ylabel("Optimal Gaussian Bandwidth")
```

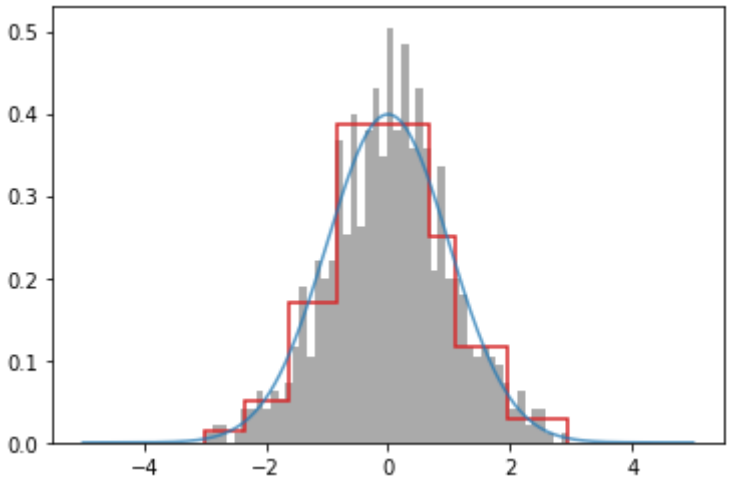


We observe the following:

- Just as in the case of the optimal binsize, the optimal Gaussian bandwidth decreases as a function of NN.
- The graph looks more noisy. This is due to the fact that 1) We use cross-validation to determine the optimal Gaussian bandwidth and 2) we evaluate the optimal Gaussian bandwidth for many more choices of NN.
- Overall, the observation aligns well with the observation above.

Additionally, we note that we can use Bayesian Blocks in the case that Δ does not need to be constant. We implement this below.

In [4]:
```
from astropy.stats import bayesian_blocks

x_data = np.random.normal(loc = 0, scale = 1, size = N)
x_grid = np.linspace(-5, 5, num=1000)
plt.plot(x_grid, norm.pdf(x_grid), marker='', alpha=0.75, zorder=10, col
or='tab:blue');
plt.hist(x_data, bins=50, histtype='stepfilled', density=True, color='#a
aaaaa');

# plot an adaptive-width histogram on top
plt.hist(x_data, bins=bayesian_blocks(x_data), color='tab:red', lw=1.5,
histtype='step', density=True);
```



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, you 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 (https://github.com/fred3m/hyperspectral/blob/master/data/subset.npy) hyper-spectral data set, taken from an airplane (https://doi.org/10.1117/12.157055) 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 Maps/Earth could turn out to be useful.*

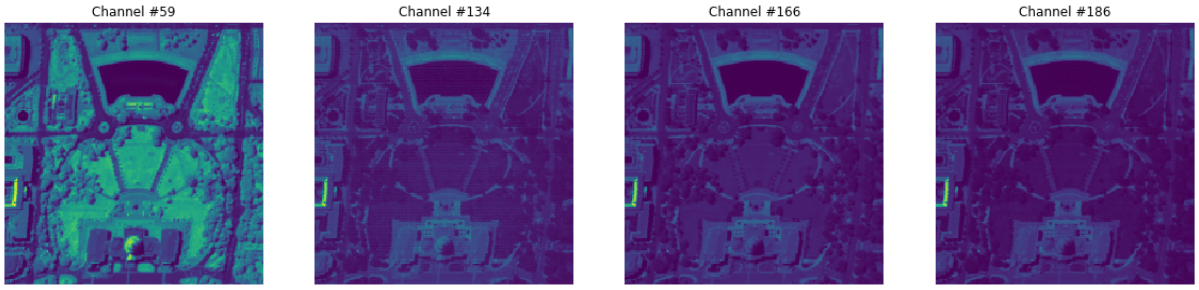
In [5]:
```
from sklearn.mixture import GaussianMixture
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import MeanShift, estimate_bandwidth
```

Load the data

In [6]:
```
data = np.load('subset.npy')

# Reshape the dataset as 191 images of size 200x200
imgs = data.reshape(191, 200, 200)
```

Visualise images for 4 channels

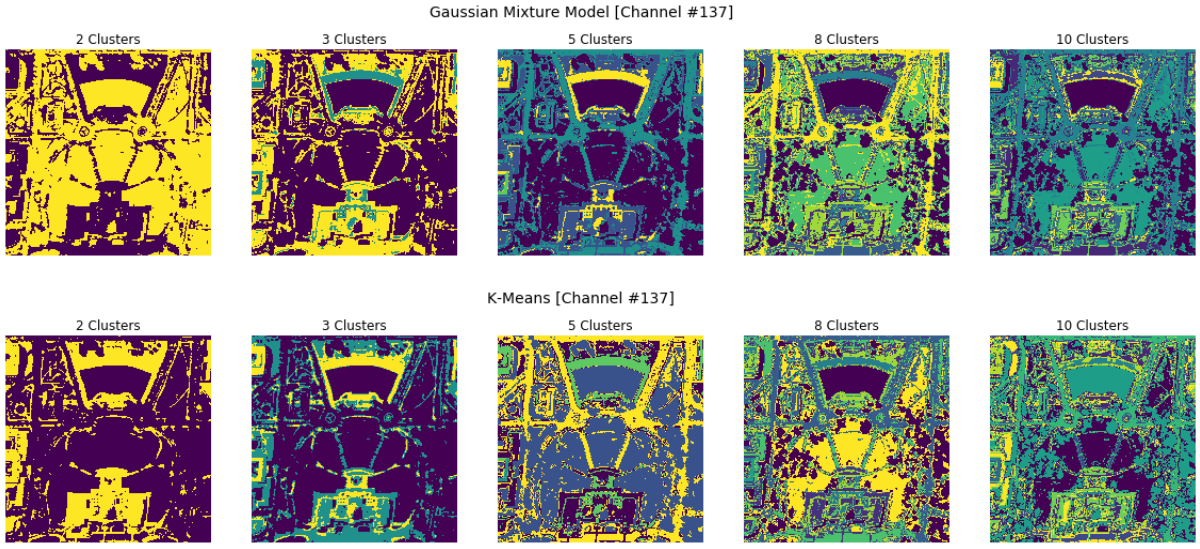
In [7]:
```
n = 4
img_nums = np.random.randint(0, imgs.shape[0], n)
fig, axes = plt.subplots(1, n, figsize=(20,5))
for i,ax in zip(img_nums, axes.flat):
    ax.imshow(imgs[i,:,:])
    ax.set_title('Channel #%s' %i)
    ax.axis('off')
plt.show()
```



Gaussian Mixture Model and K-Means Clustering

We apply a Gaussian Mixture Model and K-Means (with different numbers of clusters) to an image from a single channel to identify features and structures.

In [8]:
```
# Choose an image on which to run clustering
channel = np.random.randint(0,191)
X = data[channel,:].reshape(40000,1)

# Select the numbers of clusters to try
K = [2,3,5,8,10]

# Cluster using GMM
fig, axes = plt.subplots(1, len(K), figsize=(20,4))
for k, ax in zip(K, axes.flat):
    gmm = GaussianMixture(n_components=k, covariance_type='full')
    labels = gmm.fit_predict(X).reshape(200,200)
    ax.imshow(labels)
    ax.set_title('%s Clusters' %k)
    ax.axis('off')
fig.suptitle('Gaussian Mixture Model [Channel #%s]' %channel, fontsize=1
4)
plt.show()

# Cluster using K-Means
fig, axes = plt.subplots(1, len(K), figsize=(20,4))
for k, ax in zip(K, axes.flat):
    km = KMeans(n_clusters=k)
    labels = km.fit_predict(X).reshape(200,200)
    ax.imshow(labels)
    ax.set_title('%s Clusters' %k)
    ax.axis('off')
fig.suptitle('K-Means [Channel #%s]' %channel, fontsize=14)
plt.show()
```



We make the following observations about the process

- Parsimony seems to be beneficial in selecting the number of clusters for both GMM and K-Means. 5 or 8 or fewer clusters offer easier identification of separate structures in the images.
- Since K-Means is a special case of GMM, only using the means and assuming the covariances are diagonal, it is not too surprising that the clusters look fairly similar.
- With 2 clusters, both GMM and K-Means identify what appear to be areas with structures or roads & pavements and areas with grass or bare ground.
- Starting at 3 clusters, we begin to see some of the differences between GMM and K-Means; GMM has smoother and less noisy edges to the clusters, while K-Means is more likely to have rougher edges between clusters. This makes sense given the theory: GMM is "soft clustering" whereas K-Means is "hard clustering" (only finding centers of clusters without worrying about variance like in GMM). Furthermore, GMM is probably "smoother" since it is trying to estimate more parameters.
- At 5 or 8 clusters, the clusters start to identify differences between types of structures and features like water vs stone, etc.
- In terms of physical structures, we clearly see the Capitol Building at the bottom centre with two traffic circles in the middle of the image. The large swaths are grassy areas, while the lines around the edges and diagonally reaching the circles are roads. We can also confirm that the trapezoidal shape in the upper centre area (bright yellowy-green in the 8-cluster GMM and K-Means images) is the water feature in front of the Ulysses S. Grant Memorial.

Mean-Shift with PCA for dimensionality reduction

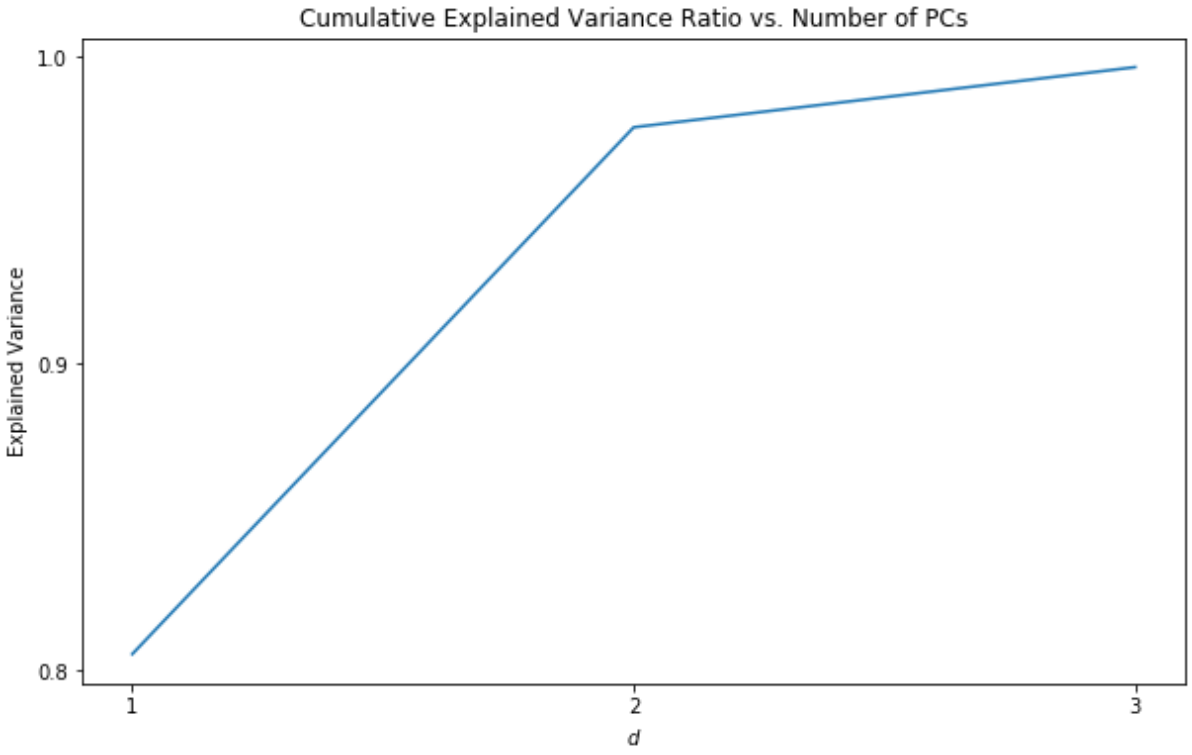
We apply Mean-Shift in combination with PCA for dimensionality reduction. We apply Mean-Shift on the PCA-projected dataset in order to overcome the curse of dimensionality. That is, each observation (which is in our case a picture) has $200 \times 200 = 40000$ dimensions (one corresponding to each pixel). Therefore, we observe that the data is very high dimensional. Mean-Shift uses kernel density estimators, which only work reliably and stably for low dimensional datasets. Therefore, we use PCA to reduce the 40000 dimensions to $d = 3$ dimensions. Thus, the underlying assumption is that the first 3 dimensions explain enough of the sample variance.

Standardise the data

In [9]:
```
scaler = StandardScaler(with_mean=True, with_std=True)
data_std = scaler.fit_transform(data)
# data_std = scaler.fit_transform(data.T).T
```

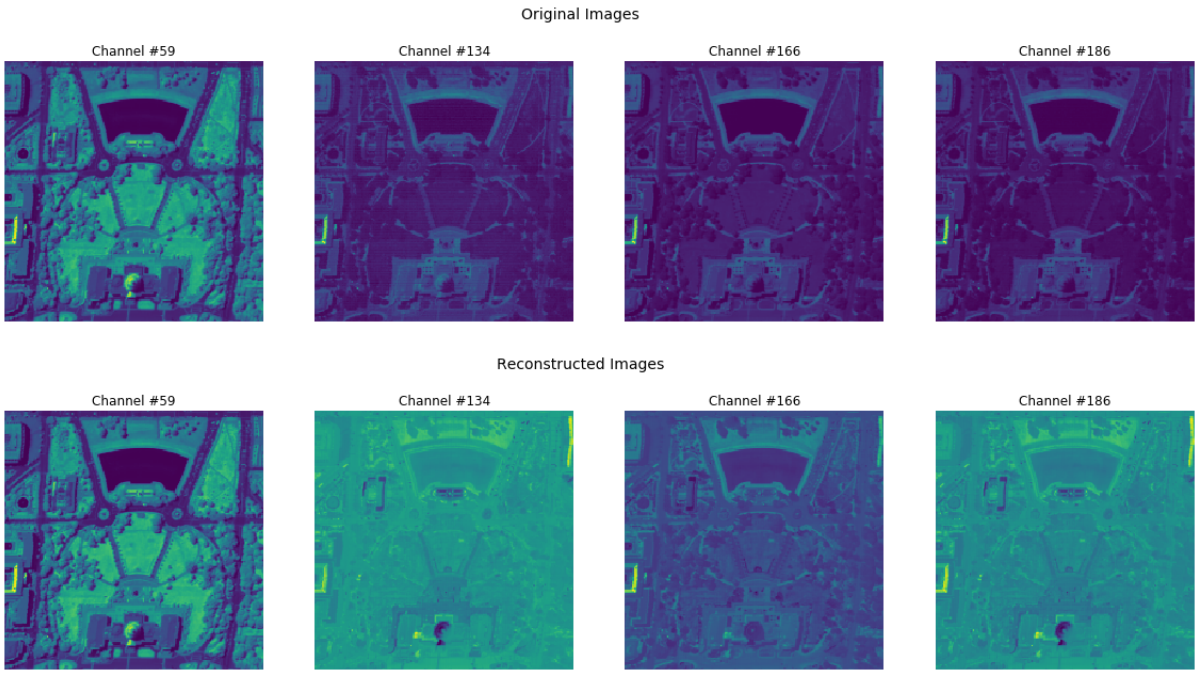
Apply PCA

In [10]:
```
# Select the number of PCs to use
num_PCs = 3
pca = PCA(n_components=num_PCs)
PCs = pca.fit_transform(data_std)
# PCs = pca.fit_transform(data_std.T)

# Reconstruct the images from just the 3 PCs
data_proj = scaler.inverse_transform(pca.inverse_transform(PCs))
# data_proj = scaler.inverse_transform(pca.inverse_transform(PCs)).T

# Check the cumulative explained variance ratio to confirm the 3 PCs are
adequate
plt.figure(figsize=(10,6))
plt.plot(range(1,4), np.cumsum(pca.explained_variance_ratio_))
plt.xticks(range(1,4))
plt.xlabel('$d$')
plt.yticks(np.arange(0.8, 1.09, 0.1))
plt.ylabel('Explained Variance')
plt.title('Cumulative Explained Variance Ratio vs. Number of PCs')
plt.show()
```



We observe that the 3 PCs explain almost all of the in-sample variance of the dataset. This indicates that most of the information of the full 200×200 pixel data set can be explained by only 3 dimensions. Now, we compare a couple of original pictures with their PCA reconstructions.

In [11]:
```
# Original images
fig, axes = plt.subplots(1, n, figsize=(20,5))
for i,ax in zip(img_nums, axes.flat):
    ax.imshow(imgs[i,:,:])
    ax.set_title('Channel #%s' %i)
    ax.axis('off')
fig.suptitle('Original Images', fontsize=14)
plt.show()

# Reconstructed images
imgs_proj = data_proj.reshape(191, 200, 200)

fig, axes = plt.subplots(1, n, figsize=(20,5))
for i,ax in zip(img_nums, axes.flat):
    ax.imshow(imgs_proj[i,:,:])
    ax.set_title('Channel #%s' %i)
    ax.axis('off')
fig.suptitle('Reconstructed Images', fontsize=14)
plt.show()
```



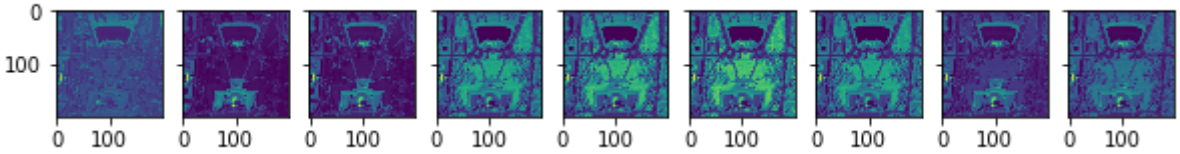
We observe that, for these random channels, the PCA reconstruction is very similar to the original images. This and the explained variance plot together suggest that PCA with 3 dimensions works well on this dataset and captures most of the information.

Apply Mean-Shift

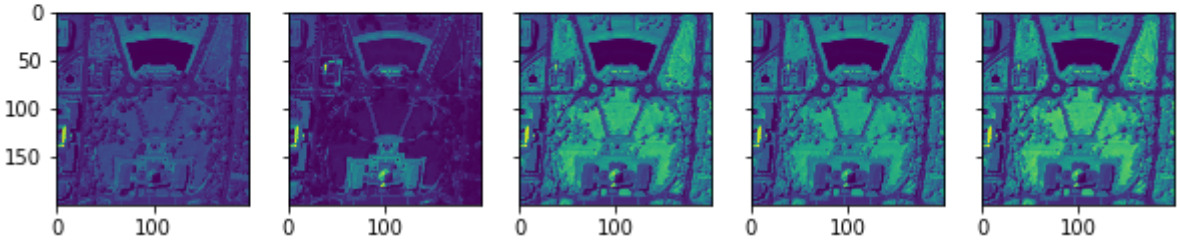
In [12]:

```
est_bandwidth = estimate_bandwidth(PCs)
bs= np.linspace(0.5*est_bandwidth, 2.0*est_bandwidth, 5)
N_centers = np.zeros(len(bs))

for i in range(len(bs)):

    ms = MeanShift(bandwidth=bs[i])
    ms.fit(PCs)
    centers = scaler.inverse_transform(pca.inverse_transform(ms.cluster_
centers_))
    N_centers[i] = int(centers.shape[0])
    fig, axes = plt.subplots(1, int(N_centers[i]), figsize=(10, 4), shar
ex=True, sharey=True)
    for j,ax in enumerate(axes):
        ax.imshow(centers[j,:].reshape((200,200)))
    fig.suptitle('PCA/Meanshift Clusters for b= '+str(round(bs[i],2)), f
ontsize=14)
    plt.show()
```

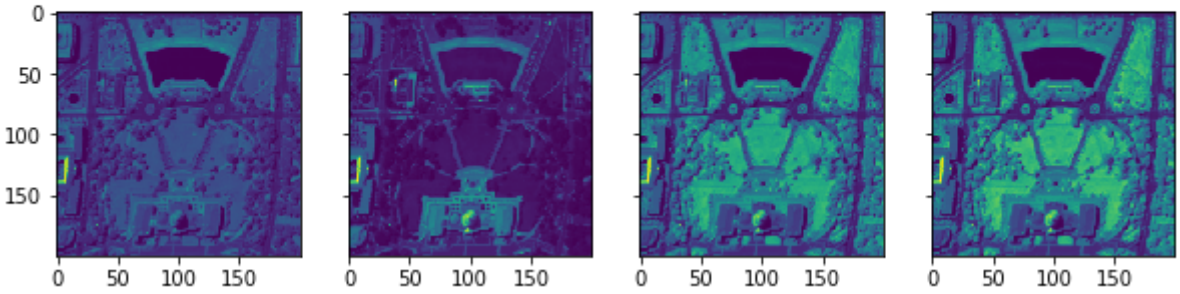
PCA/Meanshift Clusters for b= 59.99



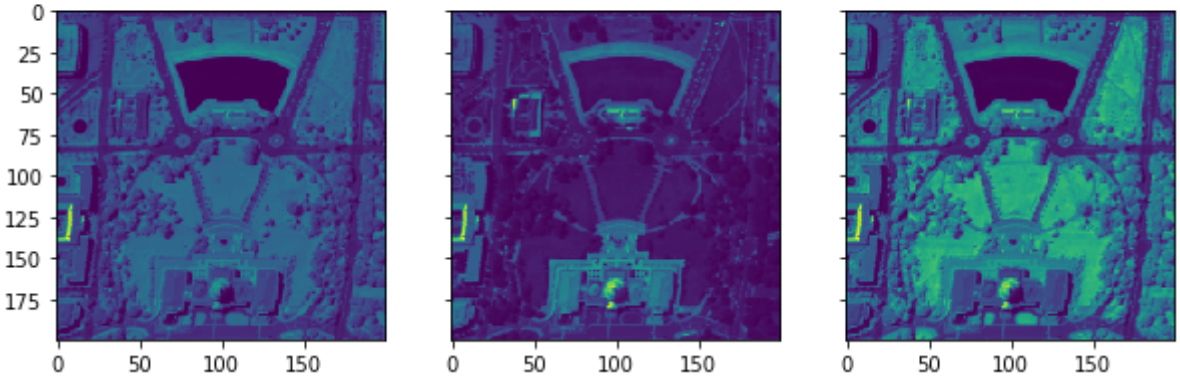
PCA/Meanshift Clusters for b= 104.98



PCA/Meanshift Clusters for b= 149.97



PCA/Meanshift Clusters for b= 194.96



The above images are representative for the different channels. Thus, we do not cluster within an image and cannot identify individual structures. However, it does allow us to practically reduce the dimension of the images and represent all the channels in a much smaller set of images.

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.

Program an EM algorithm to solve for the posterior weights π_k and mean probabilities μ_{jk} of each question, assuming these priors: $p(\pi_G) = 0.8$, $p(\pi_B) = 0.1$, $p(\pi_U) = 0.1$, $p(\mu_{jk}) \propto 1 \forall j, k$.

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 in G.*

Hint: *Murphy 11.2.2 and Exercise 11.3*

In the E-step, we calculate the responsibilities r_{ik} (taken from lecture slides):

$$r_{ik} = \frac{\pi_k \prod_j \mu_{jk}^{x_{ij}} (1-\mu_{jk})^{1-x_{ij}}}{\sum_k \pi_k \prod_j \mu_{jk}^{x_{ij}} (1-\mu_{jk})^{1-x_{ij}}}$$

and the log-likelihood is given as

$$\log L = \log \prod_i \sum_k \pi_k \prod_j \mu_{jk}^{x_{ij}} (1-\mu_{jk})^{1-x_{ij}}$$

In the M-step, we then update the parameter estimates that maximize the auxillary function $Q(\theta, \theta^t)$, adjusted with their priors (taken from Murphy 11.2.4):

$$\theta^t = \arg\max_\theta Q(\theta, \theta^{t-1}) + \log p(\theta)$$

$$\pi_k \leftarrow \frac{1}{N} \sum_i r_{ik}$$

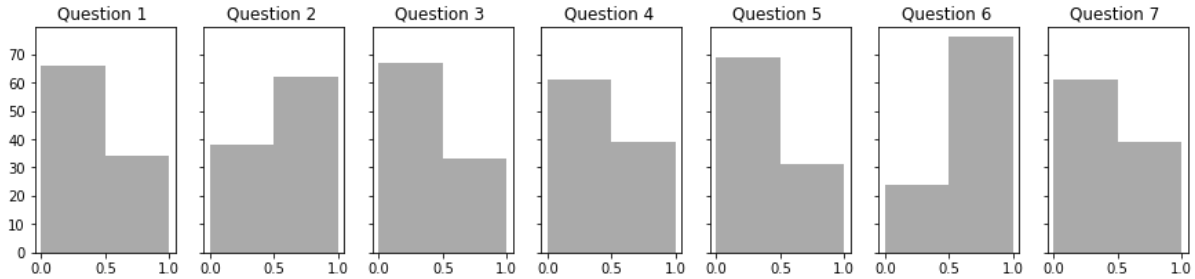
$$\mu_{jk} \leftarrow \frac{1}{\sum_i r_{ik}} \sum_i r_{ik} x_{ij}$$

Since the priors are constant with respect to θ, θ^t is updated just as usual by maximizing $Q(\theta, \theta^{t-1})$. The algorithm will stop once when the log-likelihood doesn't increase by much ($< 10^{-6}$). The EM is run with initializations for π and μ as random numbers between 0 and 1.

In [13]:
```
# load data and check the number of dimensions
questionnaire = np.load('questionnaire.npy')
questionnaire.shape
```

Out[13]:
```
(100, 7)
```

In [14]:
```
# plot histograms of data
bins = np.linspace(0, 1, 3)
fig, axes = plt.subplots(1, 7, figsize=(15, 3), sharex=True, sharey=True
)
for i in range(7):
    axes[i].hist(questionnaire[:, i], bins=bins, density=False, color='#
aaaaaa')
    axes[i].set_title('Question {0} '.format(i+1))
```



In [15]:
```
# compute e-step: in particular, compute r_ik, given some mu_jk, pi_k, a
nd data
def e_step(pi_k, mu, x):
    # mu is mu_jk (7x3)
    # x is x_ij (in our case, 100x7)
    # r_ik is 100x3
    N = x.shape[0]
    N_dist = len(pi_k)
    N_dim = mu.shape[0]


    # compute latent variables
    r_ik = np.zeros((N, N_dist))
    for i in range(N):
        # compute denominator and product in numerator
        sum_denom = 0
        factor = np.ones(N_dist)
        for k in range(N_dist):
            for j in range(N_dim):
                factor[k] = factor[k]*(mu[j][k]**(x[i][j]))*((1-mu[j][k
])**(1-x[i][j]))
            sum_denom += pi_k[k]*factor[k]

        for k in range(N_dist):
            r_ik[i][k] = pi_k[k]*factor[k]/sum_denom
    return r_ik
```

In [16]:
```
# compute m-step: in particular, update mu_jk and pi_k, given r_ik, dat
a, and priors
# here we assume Dirichlet prior for the three categories
def m_step(r_ik, x):
    N = x.shape[0]
    priors = np.array([0.8, 0.1, 0.1])
    # compute mu_jk
    sum_r_ik = np.sum(r_ik, axis = 0) # now we can index it by k (1x3)
    mu_jk = np.transpose(x)@r_ik #(7x3)
    mu_jk = mu_jk/sum_r_ik #(7x3)

    # compute pi_jk
    pi_k = (sum_r_ik + priors - 1)/(N + np.sum(priors) - 3) #1x3

    return mu_jk, pi_k
```

In [17]:
```
# compute the auxilary function Q
def compute_Q_check(r_ik, pi_k, mu_jk, x):
    # mu_jk is 7x3
    # pi_k is 1x3
    # r_ik is 100x3
    # x is 100x7
    N = x.shape[0]
    N_dist = len(pi_k)
    N_dim = mu_jk.shape[0]
    #print(N, N_dist, N_dim)
    Q = 0
    for i in range(N):
        sum_temp_i = 0
        for k in range(N_dist):
            sum_temp = 0
            for j in range(N_dim):
                sum_temp += x[i][j]*np.log(mu_jk[j][k]) + (1 - x[i][j])*
np.log(1 - mu_jk[j][k])
            sum_temp_i += r_ik[i][k]*sum_temp
        Q += sum_temp_i
    return Q
```

In [18]:
```
# combine all previous functions together into one algorithm
def EM_algorithm(pi_k_init, mu_jk_init, x, tol):
    # mu_jk is 7x3
    # pi_k is 1x3
    # r_ik is 100x3
    # x is 100x7
    N = x.shape[0]
    N_dist = len(pi_k_init)
    N_dim = mu_jk_init.shape[0]

    r_ik = e_step(pi_k_init, mu_jk_init, x)
    mu_jk, pi_k = m_step(r_ik, x)
    Q_init = compute_Q_check(r_ik, pi_k, mu_jk, x)

    # do one more step
    r_ik = e_step(pi_k, mu_jk, x)
    mu_jk, pi_k = m_step(r_ik, x)
    Q_new = compute_Q_check(r_ik, pi_k, mu_jk, x)

    #print("Q_old_0:" +str(Q_init))
    #print("Q_new_0:" +str(Q_new))
    #print("Diff: " + str(Q_new - Q_init))

    n=1
    while abs(Q_new - Q_init) > tol:
        n += 1
        Q_init = Q_new
        r_ik_new = e_step(pi_k, mu_jk, x)
        mu_jk_new, pi_k_new = m_step(r_ik_new, x)
        Q_new = compute_Q_check(r_ik_new, pi_k_new, mu_jk_new, x)

        r_ik = r_ik_new
        mu_jk, pi_k = mu_jk_new, pi_k_new

        # debugging
        #print("Q_old:" +str(Q_init))
        #print("Q_new:" +str(Q_new))
        #print(pi_k)
        #print(mu_jk)
    #print(n)
    return pi_k, mu_jk, n
```

In [19]:
```
# initialize pi_k and mu_jk to some random values
# and run the algorithm
# just to test it
questionnaire = np.load('questionnaire.npy')
pi_k_init = np.random.rand(3)
pi_k_init = pi_k_init/np.sum(pi_k_init)
mu_jk_init = np.random.rand(questionnaire.shape[1], 3)
pi_k, mu_jk, n = EM_algorithm(pi_k_init, mu_jk_init, questionnaire, 1)
n
```

Out[19]: 6

In [20]:
```
# in order to get the expectation value and the variance for the good gr
oup
# run the algorithm N times and record data only for the good group
N = 200
pi_k_tot = np.zeros(3)
mu_jk_tot = np.zeros((7, 3))
mu_jk_good = np.ones(7)
for n_temp in range(N):
    pi_k_init = np.random.rand(3)
    pi_k_init = pi_k_init/np.sum(pi_k_init)
    mu_jk_init = np.random.rand(questionnaire.shape[1], 3)
    pi_k, mu_jk, n = EM_algorithm(pi_k_init, mu_jk_init, questionnaire,
1)
    pi_k_tot += pi_k
    mu_jk_tot += mu_jk
    if n_temp == 0:
        mu_jk_good = np.vstack((np.transpose(mu_jk_good), np.transpose(m
u_jk[:, 0])))
    else:
        mu_jk_good = np.concatenate((mu_jk_good, (mu_jk[:, 0]).reshape(1
, 7)), axis = 0)
    #np.concatenate((ll, (mu_jk[:, 0]).reshape(1, 7)), axis = 0)
pi_k_tot = pi_k_tot/N
mu_jk_tot = mu_jk_tot/N
mu_jk_good = mu_jk_good[1:]
print('pi: ', pi_k_tot)
print('mu_total: ', mu_jk_tot)
print('mu_good: ', mu_jk_good)
```

```
pi:  [0.35597211 0.32015769 0.3238702 ]
mu_total:  [[0.39240633 0.34620272 0.40751293]
 [0.55887323 0.52082436 0.50383447]
 [0.40538666 0.4131121  0.42022451]
 [0.52084398 0.54436848 0.556875  ]
 [0.42632513 0.42897518 0.43428483]
 [0.70801422 0.69299515 0.66561993]
 [0.49793128 0.50257356 0.51127148]]
mu_good:  [[0.1001314  0.44282572 0.18423552 ... 0.12165158 0.84800311
0.32561895]
 [0.61649631 0.31462322 0.69844286 ... 0.9080668  0.5216171  0.8958833
9]
 [0.22891265 0.86657916 0.13404421 ... 0.02576561 0.88714773 0.0684762
1]
 ...
 [0.06361603 0.46732308 0.0530759  ... 0.11163767 0.94864244 0.3077481
9]
 [0.21491861 0.76223007 0.12669952 ... 0.05254115 0.90149551 0.1631644
8]
 [0.62825824 0.32745417 0.71774207 ... 0.94715791 0.51487008 0.9010472
9]]
```

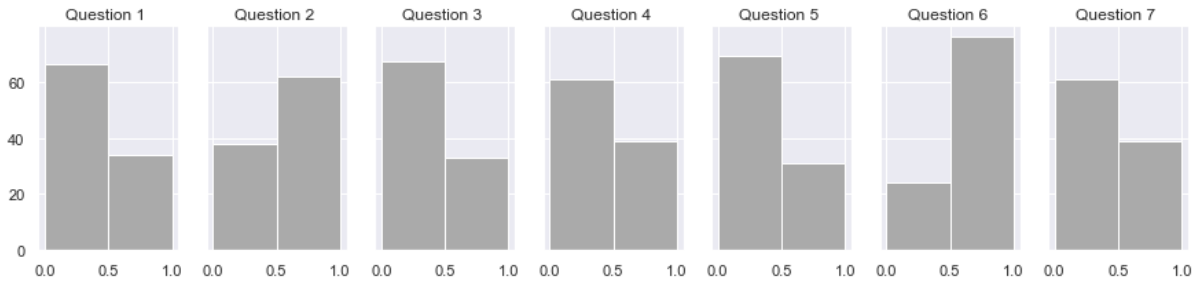
In [21]:
```
# print mean and variances for questions in group G
E_j = np.zeros(mu_jk_good.shape[1])
var_j = np.zeros(mu_jk_good.shape[1])
for i in range(mu_jk_good.shape[1]):
    E_j[i] = np.mean(mu_jk_good[:, i])
    var_j[i] = np.var(mu_jk_good[:, i])
print(E_j)
print(var_j)
```

```
[0.39240633 0.55887323 0.40538666 0.52084398 0.42632513 0.70801422
 0.49793128]
[0.06278533 0.06525624 0.10246674 0.18763532 0.17345758 0.03671573
 0.13104439]
```

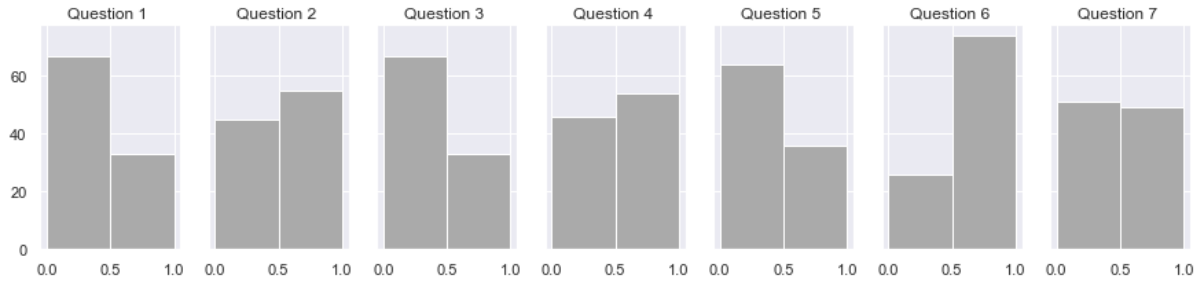
Visualise results and compare

In [22]:
```
from scipy.stats import bernoulli
import seaborn as sns
# settings for seaborn plotting style
sns.set(color_codes=True)
# settings for seaborn plot sizes
sns.set(rc={'figure.figsize':(5,5)})
```

In [23]:
```
# questionnaire data
fig, axes = plt.subplots(1, 7, figsize=(15, 3), sharex=True, sharey=True
)
for i in range(7):
    axes[i].hist(questionnaire[:, i], bins=bins, density=False, color='#
aaaaaa')
    axes[i].set_title('Question {0} '.format(i+1))
```



In [24]:
```
# mixture of distributions
# using mean values of N = 200 runs
fig, axes = plt.subplots(1, 7, figsize=(15, 3), sharex=True, sharey=True
)
mu_jk = mu_jk_tot
pi_k = pi_k_tot
for i in range(7):
    data_bern= pi_k[0]*bernoulli.rvs(size=100,p=mu_jk[i][0]) + pi_k[1]*
bernoulli.rvs(size=100,p=mu_jk[i][1]) +pi_k[2]*bernoulli.rvs(size=100,p=
mu_jk[i][2])
    axes[i].hist(data_bern, bins=bins, density=False, color='#aaaaaa')
    axes[i].set_title('Question {0} '.format(i+1))
```



Step 2 (1pts)

Can you construct a more precise estimator of μ_j given the responses from all users?

Hint: *Realize that the mean and variance over all users is analytic for this mixture model. However, not every user has the same behavior and motivation. If you want to reject some users, which ones would you chose?*

Comments

Ideally, we would want to reject the bad users in group B. Because they are aren't paying attention to their answers, they are not giving an intentional correct or incorrect response. Even with ugly users in group U, we can at least parse out information about their response patterns, and try to distinguish that from the good users in G. The bad users, since they are unpredictable, simply add noise to the estimation of response patterns across three groups, since they could randomly respond like one of the two other groups or do something completely different. Thus, removing these users (if possible) would definitely allow for more precision estimation of each μ_j.