TrachtPS5

February 7, 2019

1 Problem Set 5

1.0.1 by Daniel Tracht, February 2019

1.1 Problem 1

In this problem, we will be using the similar (though to my surprise not the same) data on incomes of graduates from the MACSS program as in the previous data set. We will also be using the log normal distribution as before.

1.1.1 Part a

We want to plot a histogram of the income data:

```
In [1]: # Import the necessary libraries
        # Import packages and load the data
        import numpy as np
        import numpy.linalg as lin
        import scipy.stats as sts
        import scipy.integrate as intgr
        import scipy.optimize as opt

        import requests

        # Download and save the data file incomes.txt
        url = ('https://raw.githubusercontent.com/dtracht/persp-model-econ_W19/' +
               'master/ProblemSets/PS5/data/incomes.txt')
        data_file = requests.get(url, allow_redirects=True)
        open('data/incomes.txt', 'wb').write(data_file.content)

        # Load the data as a NumPy array
        pts = np.loadtxt('data/incomes.txt')

In [2]: import matplotlib.pyplot as plt

        num_bins = 30
        # normed option has been deprecated
        # using density=True instead
```

```
count, bins, ignored = plt.hist(pts, num_bins, density=True,
                                edgecolor='k')
plt.title("MACSS Incomes, Classes of 2018-20", fontsize=20)
plt.xlabel("Incomes in U.S. Dollars")
plt.ylabel("Percent of incomes")
plt.xlim([0,140000])

plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

1.1.2 Part b

We want to estimate the parameters of the log normal distribution through generalized method of moments estimation. We will use the average income and standard deviation of income as our two moments. We will use the identity matrix as our weighting matrix.

In [3]:
```
# takes vector of x values, parameters for mu and sigma
# returns vector of pdf values for the lognormal
from scipy.stats import lognorm
def lognormal_pdf (xvals, mu, sigma):
    pdf_vals = (1 / (xvals * sigma * np.sqrt(2 * np.pi))) * np.exp (-(np.log(xvals)-mu
    return pdf_vals

# takes vector of data
# returns mean and variance of data
def data_moments(xvals):
    mean_data = xvals.mean()
    var_data = xvals.var()
    return mean_data, var_data

# takes parameters for mu and sigma
# returns approximate mean and variance for lognormal with given parameters
def model_moments(mu, sigma):
    xfx = lambda x: x * lognormal_pdf(x, mu, sigma)
    # integrating over to 3 orders more than the upper bound of the data
    (mean_model, m_m_err) = intgr.quad(xfx, 1e-10, 1.4e8)
    x2fx = lambda x: ((x - mean_model) ** 2) * lognormal_pdf(x, mu, sigma)
    (var_model, v_m_err) = intgr.quad(x2fx, 1e-10, 1.4e8)
    return mean_model, var_model

# takes vector of data, parameters for mu and sigma, and boolean to use raw change or p
# boolean defaulted to false, that is uses percent change by default
# returns error vector between the model moments and data moments
def err_vec(xvals, mu, sigma, simple=False):
    mean_data, var_data = data_moments(xvals)
    moms_data = np.array([[mean_data], [var_data]])
```

```
        mean_model, var_model = model_moments(mu, sigma)
        moms_model = np.array([[mean_model], [var_model]])
        if simple:
            err_vec = moms_model - moms_data
        else:
            err_vec = (moms_model - moms_data) / moms_data
        return err_vec

    # takes vector of parameters to estimate
    # takes tuple of arguments with data and weight matrix
    # returns criterion value using error vector and weight matrix
    def criterion(params, *args):
        mu, sigma = params
        xvals, W = args
        err = err_vec(xvals, mu, sigma, simple=False)
        crit_val = err.T @ W @ err
        return crit_val
```

```
In [4]: import scipy.optimize as opt

        # start with initial guess from previoius assignment
        mu_init = 11
        sig_init = 0.5
        params_init = np.array([mu_init, sig_init])
        # using identity matrix for weighting
        W_hat1 = np.eye(2)
        gmm_args = (pts, W_hat1)
        results = opt.minimize(criterion, params_init, args=(gmm_args), tol=1e-14,
                               method='L-BFGS-B', bounds=((1e-10, None), (1e-10, None)))
        mu_GMM1, sig_GMM1 = results.x
        print('mu_GMM1=', mu_GMM1, ' sig_GMM1=', sig_GMM1)
```

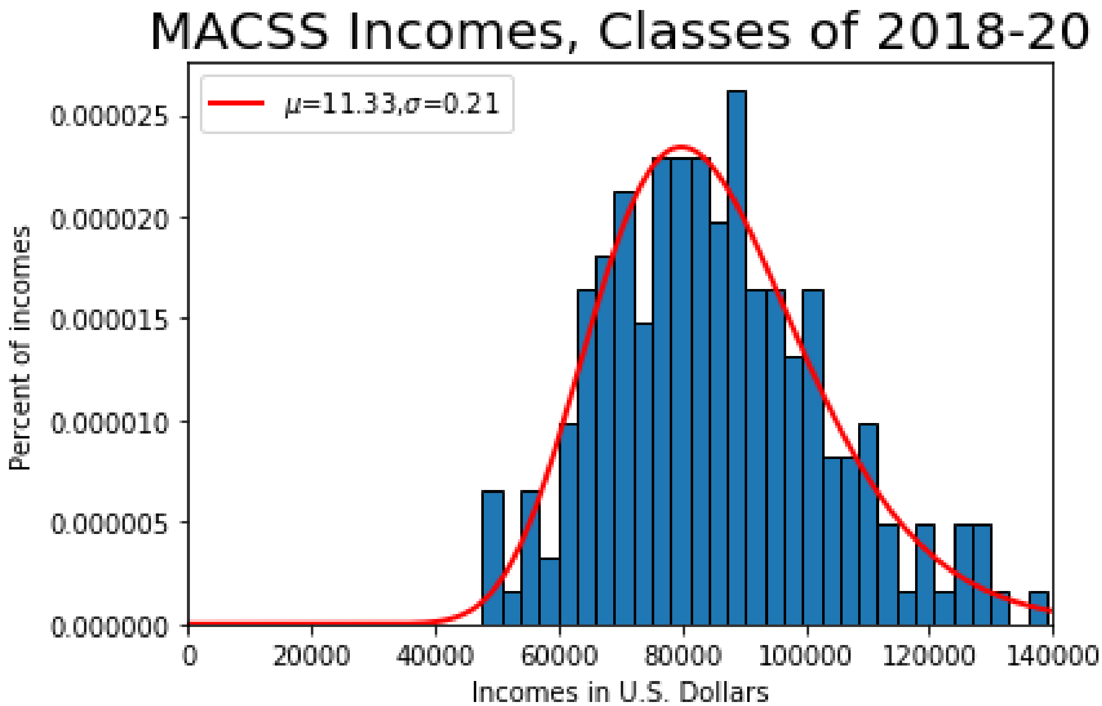
```
mu_GMM1= 11.331880928760944  sig_GMM1= 0.20869663980785672
```

We wish to plot the estimated lognormal PDF against the histogram in part a:

```
In [5]: # Plot histogram with the estimated PDF
        num_bins = 30
        # normed option has been deprecated
        # using density=True instead
        count, bins, ignored = plt.hist(pts, num_bins, density=True,
                                        edgecolor='k')
        plt.title("MACSS Incomes, Classes of 2018-20", fontsize=20)
        plt.xlabel("Incomes in U.S. Dollars")
        plt.ylabel("Percent of incomes")
        plt.xlim([0,140000])
```

```
dist_pts = np.linspace(1e-10, 140000, 1000)
plt.plot(dist_pts, lognormal_pdf(dist_pts, mu_GMM1, sig_GMM1),
         linewidth=2, color='r', label='$\mu$=11.33,$\sigma$=0.21')
plt.legend(loc='upper left')

plt.show()
```



We want the value of the criterion function at the estimated GMM parameters. We also want to compare the data moments to the model moments at the estimated parameter values:

```
In [6]: mean_data, var_data = data_moments(pts)
        mean_model, var_model = model_moments(mu_GMM1, sig_GMM1)
        err1 = err_vec(pts, mu_GMM1, sig_GMM1, False).reshape(2,)
        crit_val = err1.T @ W_hat1 @ err1
        print('Mean of points =', mean_data, ', Variance of points =', var_data)
        print('Mean of model =', mean_model, ', Variance of model =', var_model)
        print('Error vector=', err1)
        print("Criterion value at estimated parameters=", crit_val)
```

```
Mean of points = 85276.82360625811 , Variance of points = 323731572.2295289
Mean of model = 85276.83046522719 , Variance of model = 323731555.17099494
Error vector= [ 8.04318077e-08 -5.26934517e-08]
Criterion value at estimated parameters= 9.24587554357195e-15
```

1.1.3 Part c

We wish to perform a two-step GMM procedure by using the estimates from above to generate an estimator for the variance covariance matrix

$$\hat{\Omega}_{2step}$$

which we then use to get the optimal weighting matrix

$$\hat{W}_{2step}:$$

In [7]:
```
# takes vector of data, parameters for mu and sigma, and boolean as in err_vec functio
# returns matrix of variance and covariance between the model moments
def get_Err_mat2(pts, mu, sigma, simple=False):
    R = 2
    N = len(pts)
    Err_mat = np.zeros((R, N))
    mean_model, var_model = model_moments(mu, sigma)
    if simple:
        Err_mat[0, :] = pts - mean_model
        Err_mat[1, :] = ((mean_data - pts) ** 2) - var_model
    else:
        Err_mat[0, :] = (pts - mean_model) / mean_model
        Err_mat[1, :] = (((mean_data - pts) ** 2) - var_model) / var_model
    return Err_mat
```

In [8]:
```
# starting with the results from the first step
Err_mat = get_Err_mat2(pts, mu_GMM1, sig_GMM1, False)
VCV2 = (1 / pts.shape[0]) * (Err_mat @ Err_mat.T)
print(VCV2)
# invert variance-covariance matrix to get estimated weight matrix
W_hat2 = lin.pinv(VCV2)
print(W_hat2)
```

```
[[0.0445167  0.09358616]
 [0.09358616 1.94756682]]
[[24.98774562 -1.20073267]
 [-1.20073267  0.57115984]]
```

In [9]:
```
# uses first step estimates of parameters as initial guess
params_init = np.array([mu_GMM1, sig_GMM1])
# uses same data and weight matrix from 2-step procedure
gmm_args = (pts, W_hat2)
results = opt.minimize(criterion, params_init, args=(gmm_args),
                       method='L-BFGS-B', bounds=((1e-10, None), (1e-10, None)))
mu_GMM2, sig_GMM2 = results.x
print('mu_GMM2=', mu_GMM2, ' sig_GMM2=', sig_GMM2)
```

```
mu_GMM2= 11.331880928760944  sig_GMM2= 0.20869663980785672
```

As expected, the point estimates of the parameters are the same as above. We wish to see the value of the criterion function at the estimated parameters, as well as compare the data moments to the model moments at the generated parameter:

```
In [10]: mean_data, var_data = data_moments(pts)
         mean_model, var_model = model_moments(mu_GMM2, sig_GMM2)
         err2 = err_vec(pts, mu_GMM2, sig_GMM2, False).reshape(2,)
         crit_val = err2.T @ W_hat2 @ err2
         print('Mean of points =', mean_data, ', Variance of points =', var_data)
         print('Mean of model =', mean_model, ', Variance of model =', var_model)
         print('Error vector=', err2)
         print("Criterion value at estimated parameters=", crit_val)
```

```
Mean of points = 85276.82360625811 , Variance of points = 323731572.2295289
Mean of model = 85276.83046522719 , Variance of model = 323731555.17099494
Error vector= [ 8.04318077e-08 -5.26934517e-08]
Criterion value at estimated parameters= 1.7341645910524646e-13
```

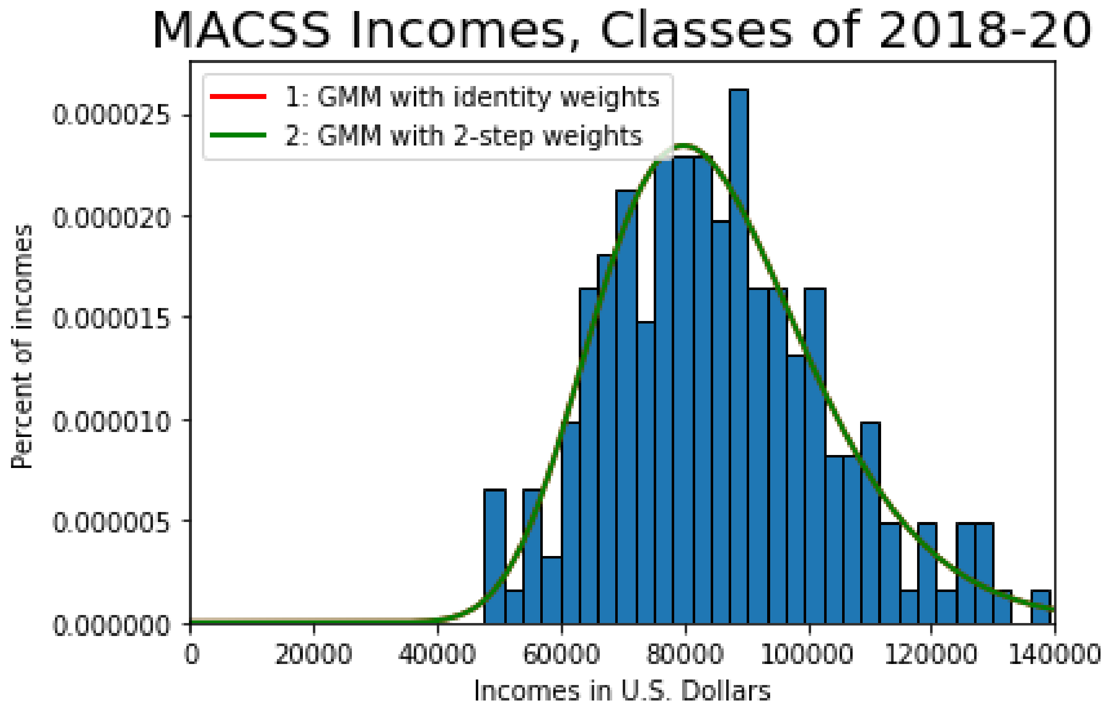
We also want to plot the PDF of the lognormal distribution for the estimated parameters as well as the histogram of the actual data and the PDF of the lognormal distribution from part b:

```
In [11]: # Plot histogram with the estimated PDF
         num_bins = 30
         # normed option has been deprecated
         # using density=True instead
         count, bins, ignored = plt.hist(pts, num_bins, density=True,
                                         edgecolor='k')
         plt.title("MACSS Incomes, Classes of 2018-20", fontsize=20)
         plt.xlabel("Incomes in U.S. Dollars")
         plt.ylabel("Percent of incomes")
         plt.xlim([0,140000])


         dist_pts = np.linspace(1e-10, 140000, 1000)
         plt.plot(dist_pts, lognormal_pdf(dist_pts, mu_GMM1, sig_GMM1),
                  linewidth=2, color='r', label='1: GMM with identity weights')
         plt.legend(loc='upper left')

         plt.plot(dist_pts, lognormal_pdf(dist_pts, mu_GMM2, sig_GMM2),
                  linewidth=2, color='g', label='2: GMM with 2-step weights')
         plt.legend(loc='upper left')


         plt.show()
```



1.1.4 Part d

We wish to estimate the lognormal PDF to fit the data using GMM on different moments. In this part, we will use the percents of indviduals earning less than 75,000 dollars, between 75 and 100 thousand dollars, and more than 100 thousand dollars:

```
In [12]: # takes vector of data
         # returns percent below 75K, between 75K and 100K, and above 100k in data
         def data_moments3(xvals):
             bpct_1_dat = xvals[xvals < 75000].shape[0] / xvals.shape[0]
             bpct_2_dat = (xvals[(xvals >=75000) & (xvals < 1.2e5)].shape[0] /
                           xvals.shape[0])
             bpct_3_dat = xvals[xvals >= 1.2e5].shape[0] / xvals.shape[0]
             return bpct_1_dat, bpct_2_dat, bpct_3_dat

         # takes parameters for mu and sigma
         # returns percent below 75k, between 75k and 100k, and above 100k in lognormal distri
         def model_moments3(mu, sigma):
             fx = lambda x: lognormal_pdf(x, mu, sigma)
             # lower bound near 0
             # upper bound 3 orders higher than top of data
             (bpct_1_mod, bp_1_err) = intgr.quad(fx, 1e-10, 7.5e4)
             (bpct_2_mod, bp_2_err) = intgr.quad(fx, 7.5e4, 1.2e5)
             (bpct_3_mod, bp_3_err) = intgr.quad(fx, 1.2e5, 1.4e8)
```

```
        return bpct_1_mod, bpct_2_mod, bpct_3_mod

    # takes vector of data, parameters of mu and sigma, and boolean as used above
    # returns error vector between model and data moments
    def err_vec3(xvals, mu, sigma, simple):
        bpct_1_dat, bpct_2_dat, bpct_3_dat = data_moments3(xvals)
        moms_data = np.array([[bpct_1_dat], [bpct_2_dat], [bpct_3_dat]])
        bpct_1_mod, bpct_2_mod, bpct_3_mod = model_moments3(mu, sigma)
        moms_model = np.array([[bpct_1_mod], [bpct_2_mod], [bpct_3_mod]])
        if simple:
            err_vec = moms_model - moms_data
        else:
            err_vec = (moms_model - moms_data) / moms_data
        return err_vec

    # takes parameter vector containing mu and sigma
    # takes tuple with vector of data and weight matrix
    # returns criterion value using error vector and weight matrix
    def criterion3(params, *args):
        mu, sigma = params
        xvals, W = args
        err = err_vec3(xvals, mu, sigma, simple=False)
        crit_val = err.T @ W @ err
        return crit_val
```

```
In [13]: # starting at parameters from previous
         mu_init = 11
         sig_init = 0.5
         params_init_3 = np.array([mu_init, sig_init])
         # using identity matrix for weights
         W_hat1_3 = np.eye(3)
         gmm_args_3 = (pts, W_hat1_3)
         results_3 = opt.minimize(criterion3, params_init_3, args=(gmm_args_3),
                                  method='L-BFGS-B', bounds=((1e-10, None), (1e-10, None)))
         mu_GMM1_3, sig_GMM1_3 = results_3.x
         print('mu_GMM1_3=', mu_GMM1_3, ' sig_GMM1_3=', sig_GMM1_3)
```

```
mu_GMM1_3= 11.33886315402999  sig_GMM1_3= 0.21666600644065456
```

We wish to see the value of the criterion function at the estimated parameters, as well as compare the data moments to the model moments at the generated parameters:

```
In [14]: bpct_1_dat, bpct_2_dat, bpct_3_dat = data_moments3(pts)
         bpct_1_mod, bpct_2_mod, bpct_3_mod = model_moments3(mu_GMM1_3, sig_GMM1_3)
         err1_3 = err_vec3(pts, mu_GMM1_3, sig_GMM1_3, False).reshape(3,)
         crit_val_3 = err1_3.T @ W_hat1_3 @ err1_3
         print('Percent lower than 75k in data =', bpct_1_dat, ', in model =', bpct_1_mod)
         print('Percent between 75k and 100k in data =', bpct_2_dat, ', in model =', bpct_2_mo
```

```
print('Percent above 100k in data =', bpct_3_dat, ', in model =', bpct_3_mod)
print('Error vector=', err1_3)
print("Criterion value at estimated parameters=", crit_val_3)
```

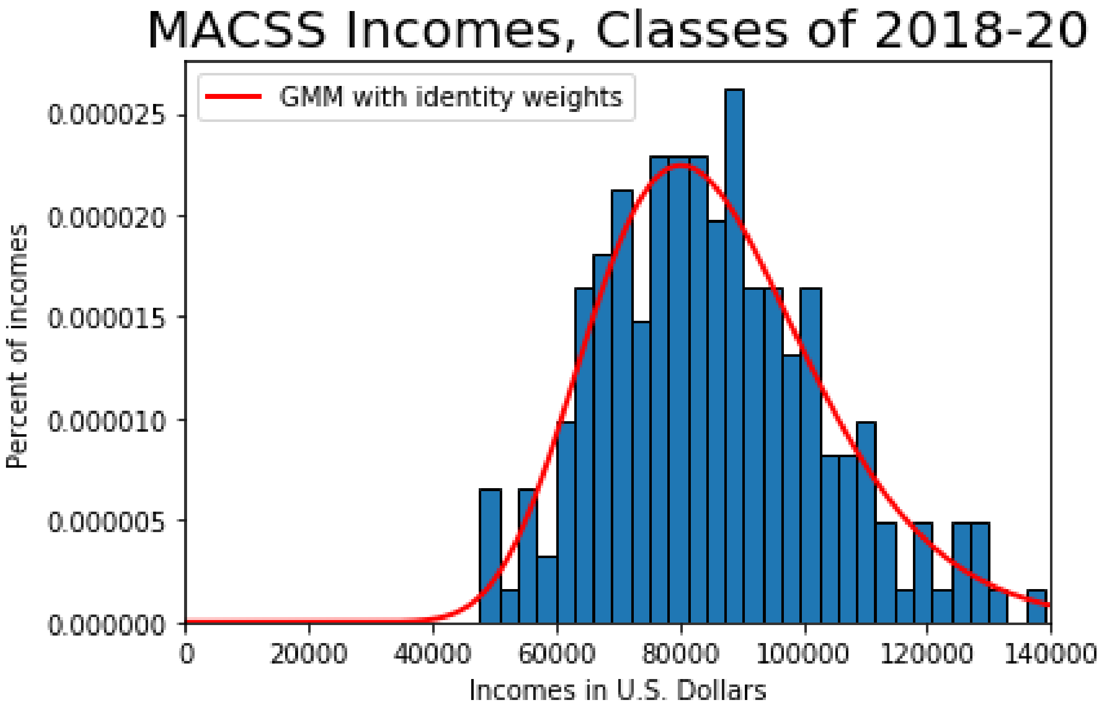
```
Percent lower than 75k in data = 0.3 , in model = 0.3000000053796349
Percent between 75k and 100k in data = 0.65 , in model = 0.649999995193099
Percent above 100k in data = 0.05 , in model = 0.04999999942726667
Error vector= [ 1.79321163e-08 -7.39523229e-09 -1.14546667e-08]
Criterion value at estimated parameters= 5.074596455991511e-16
```

We also want to plot the PDF of the lognormal distribution for the estimated parameters as well as the histogram of the actual data:

```
In [15]: # Plot histogram with the estimated PDF
         num_bins = 30
         # normed option has been deprecated
         # using density=True instead
         count, bins, ignored = plt.hist(pts, num_bins, density=True,
                                         edgecolor='k')
         plt.title("MACSS Incomes, Classes of 2018-20", fontsize=20)
         plt.xlabel("Incomes in U.S. Dollars")
         plt.ylabel("Percent of incomes")
         plt.xlim([0,140000])


         dist_pts = np.linspace(1e-10, 140000, 1000)
         plt.plot(dist_pts, lognormal_pdf(dist_pts, mu_GMM1_3, sig_GMM1_3),
                  linewidth=2, color='r', label='GMM with identity weights')
         plt.legend(loc='upper left')

         plt.show()
```



1.1.5 Part e

We wish to perform a two-step GMM procedure by using the estimates from above to generate an estimator for the variance covariance matrix

$$\hat{\Omega}_{2step}$$

which we then use to get the optimal weighting matrix

$$\hat{W}_{2step}$$

for our three moments

```
In [16]: # takes vector of data, parameters for mu and sigma, and boolean as described above
         # return error matrix
         def get_Err_mat3(pts, mu, sigma, simple=False):
             R = 3
             N = len(pts)
             Err_mat = np.zeros((R, N))
             pct_1_mod, pct_2_mod, pct_3_mod = model_moments3(mu, sigma)
             if simple:
                 pts_in_grp1 = pts < 7.5e4
                 Err_mat[0, :] = pts_in_grp1 - pct_1_mod
                 pts_in_grp2 = (pts >= 7.5e4) & (pts < 1e5)
                 Err_mat[1, :] = pts_in_grp2 - pct_2_mod
```

```
            pts_in_grp3 = pts >= 1e5
            Err_mat[2, :] = pts_in_grp3 - pct_3_mod
        else:
            pts_in_grp1 = pts < 7.5e4
            Err_mat[0, :] = (pts_in_grp1 - pct_1_mod) / pct_1_mod
            pts_in_grp2 = (pts >= 7.5e4) & (pts < 1e5)
            Err_mat[1, :] = (pts_in_grp2 - pct_2_mod) / pct_2_mod
            pts_in_grp3 = pts >= 1e5
            Err_mat[2, :] = (pts_in_grp3 - pct_3_mod) / pct_3_mod
        return Err_mat
```

```
In [17]: Err_mat3 = get_Err_mat3(pts, mu_GMM1_3, sig_GMM1_3, True)
         VCV2_3 = (1 / pts.shape[0]) * (Err_mat3 @ Err_mat3.T)
         # using the pseudoinverse here
         W_hat2_3 = lin.pinv(VCV2_3)
```

```
In [18]: # starting with the mu and sigma from above
         params_init = np.array([mu_GMM1_3, sig_GMM1_3])
         # using the 2-step weights
         gmm_args = (pts, W_hat2_3)
         results2_3 = opt.minimize(criterion3, params_init, args=(gmm_args),
                                   method='L-BFGS-B', bounds=((1e-10, None), (1e-10, None)))
         mu_GMM2_3, sig_GMM2_3 = results2_3.x
         print('mu_GMM2_3=', mu_GMM2_3, ' sig_GMM2_3=', sig_GMM2_3)
```

```
mu_GMM2_3= 11.33886315402999  sig_GMM2_3= 0.21666600644065456
```

We wish to see the value of the criterion function at the estimated parameters, as well as compare the data moments to the model moments at the generated parameters:

```
In [19]: bpct_1_dat, bpct_2_dat, bpct_3_dat = data_moments3(pts)
         bpct_1_mod, bpct_2_mod, bpct_3_mod = model_moments3(mu_GMM2_3, sig_GMM2_3)
         err2_3 = err_vec3(pts, mu_GMM2_3, sig_GMM2_3, False).reshape(3,)
         crit_val_4 = err2_3.T @ W_hat2_3 @ err2_3
         print('Percent lower than 75k in data =', bpct_1_dat, ', in model =', bpct_1_mod)
         print('Percent between 75k and 100k in data =', bpct_2_dat, ', in model =', bpct_2_mod
         print('Percent above 100k in data =', bpct_3_dat, ', in model =', bpct_3_mod)
         print('Error vector=', err2_3)
         print("Criterion value at estimated parameters=", crit_val_4)
```

```
Percent lower than 75k in data = 0.3 , in model = 0.3000000053796349
Percent between 75k and 100k in data = 0.65 , in model = 0.649999995193099
Percent above 100k in data = 0.05 , in model = 0.04999999942726667
Error vector= [ 1.79321163e-08 -7.39523229e-09 -1.14546667e-08]
Criterion value at estimated parameters= 0.0006450775382064906
```

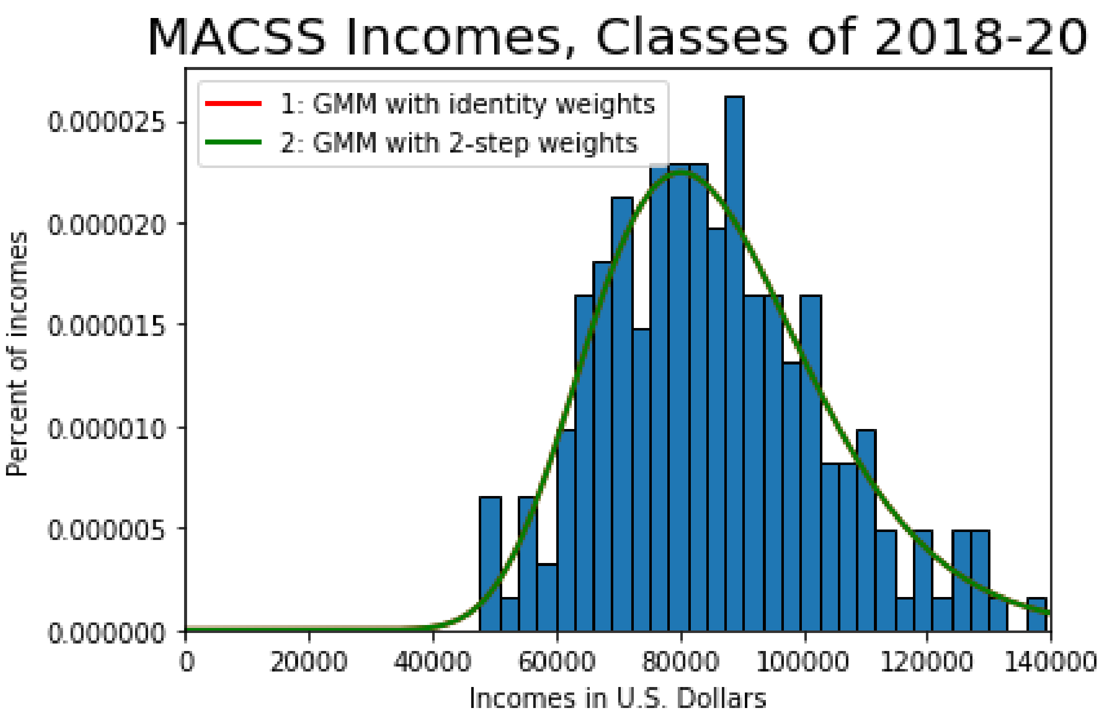
We also want to plot the PDF of the lognormal distribution for the estimated parameters as well as the histogram of the actual data and the PDF of the lognormal estimated with identity weights:

In [20]:
```python
# Plot histogram with the estimated PDF
num_bins = 30
# normed option has been deprecated
# using density=True instead
count, bins, ignored = plt.hist(pts, num_bins, density=True,
                                edgecolor='k')
plt.title("MACSS Incomes, Classes of 2018-20", fontsize=20)
plt.xlabel("Incomes in U.S. Dollars")
plt.ylabel("Percent of incomes")
plt.xlim([0,140000])


dist_pts = np.linspace(1e-10, 140000, 1000)
plt.plot(dist_pts, lognormal_pdf(dist_pts, mu_GMM1_3, sig_GMM1_3),
         linewidth=2, color='r', label='1: GMM with identity weights')
plt.legend(loc='upper left')

plt.plot(dist_pts, lognormal_pdf(dist_pts, mu_GMM2_3, sig_GMM2_3),
         linewidth=2, color='g', label='2: GMM with 2-step weights')
plt.legend(loc='upper left')


plt.show()
```



1.1.6 Part f

We wish to determine which of the four estimations fits the data the best. As we can see in our previous results, the estimated parameters are equal for all four of the estimations. As such, the likelihood of observing the data given a set of parameters is equal across the four estimations.

For "best" we might appeal to the standard errors of the estimated parameters:

```
In [21]: # takes vector of data, parameters for mu and sigma, and boolean as described above
         # returns Jacobian matrix for mu and sigma as moments
         def Jac_err2(xvals, mu, sigma, simple=False):
             Jac_err = np.zeros((2, 2))
             h_mu = 1e-8 * mu
             h_sig = 1e-8 * sigma
             Jac_err[:, 0] = \
                 ((err_vec(xvals, mu + h_mu, sigma, simple) -
                   err_vec(xvals, mu - h_mu, sigma, simple)) / (2 * h_mu)).flatten()
             Jac_err[:, 1] = \
                 ((err_vec(xvals, mu, sigma + h_sig, simple) -
                   err_vec(xvals, mu, sigma - h_sig, simple)) / (2 * h_sig)).flatten()
             return Jac_err
```

```
In [22]: # takes vector of data, parameters for mu and sigma, and boolean as described above
         # returns Jacobian matrix for three percents as moments
         def Jac_err3(xvals, mu, sigma, simple=False):
             Jac_err = np.zeros((3, 2))
             h_mu = 1e-8 * mu
             h_sig = 1e-8 * sigma
             Jac_err[:, 0] = \
                 ((err_vec3(xvals, mu + h_mu, sigma, simple) -
                   err_vec3(xvals, mu - h_mu, sigma, simple)) / (2 * h_mu)).flatten()
             Jac_err[:, 1] = \
                 ((err_vec3(xvals, mu, sigma + h_sig, simple) -
                   err_vec3(xvals, mu, sigma - h_sig, simple)) / (2 * h_sig)).flatten()
             return Jac_err
```

```
In [23]: N = len(pts)
         d_err1 = Jac_err2(pts, mu_GMM1, sig_GMM1, False)
         SigHat1 = (1 / N) * lin.inv(d_err1.T @ W_hat1 @ d_err1)
         print("For 2 moments and identity")
         print('Std. err. mu_hat=', np.sqrt(SigHat1[0, 0]))
         print('Std. err. sig_hat=', np.sqrt(SigHat1[1, 1]))
```

```
For 2 moments and identity
Std. err. mu_hat= 0.07373971821074995
Std. err. sig_hat= 0.016144777882961366
```

```
In [24]: d_err2 = Jac_err2(pts, mu_GMM2, sig_GMM2, False)
         SigHat2 = (1 / N) * lin.inv(d_err2.T @ W_hat2 @ d_err2)
```

```python
print("For 2 moments and 2-step weights:")
print('Std. err. mu_hat=', np.sqrt(SigHat2[0, 0]))
print('Std. err. sig_hat=', np.sqrt(SigHat2[1, 1]))
```

```
For 2 moments and 2-step weights:
Std. err. mu_hat= 0.01501964103731327
Std. err. sig_hat= 0.009554897368788642
```

```python
In [25]: d_err1_3 = Jac_err3(pts, mu_GMM1_3, sig_GMM1_3, False)
         SigHat1_3 = (1 / N) * lin.inv(d_err1_3.T @ W_hat1_3 @ d_err1_3)
         print("For 3 moments and identity:")
         print('Std. err. mu_hat=', np.sqrt(SigHat1_3[0, 0]))
         print('Std. err. sigma_hat=', np.sqrt(SigHat1_3[1, 1]))
```

```
For 3 moments and identity:
Std. err. mu_hat= 0.009336424949022612
Std. err. sigma_hat= 0.006410643545058238
```

```python
In [26]: d_err2_3 = Jac_err3(pts, mu_GMM2_3, sig_GMM2_3, False)
         SigHat2_3 = (1 / N) * lin.inv(d_err2_3.T @ W_hat2_3 @ d_err2_3)
         print("For 3 moments and 2-step weights:")
         print('Std. err. mu_hat=', np.sqrt(SigHat2_3[0, 0]))
         print('Std. err. sigma_hat=', np.sqrt(SigHat2_3[1, 1]))
```

```
For 3 moments and 2-step weights:
Std. err. mu_hat= 0.0047887329650079915
Std. err. sigma_hat= 0.0017719704159597101
```

As we can see, each estimation gave us lower standard errors than before. Thus, while each estimation produced identical point estimates of the parameters, the fourth technique, which used 3 moments and weights produced from the 2-step variance-covariance matrix, gave us the greatest power to reject null hypotheses.

1.2 Problem 2

In this problem, we will be estimating the parameters of a linear regression model of the form

$$sick_i = \beta_0 + \beta_1 * age_i + \beta_2 * children_i + \beta_3 * temp_winter_i + \varepsilon_i$$

1.2.1 Part a

We wish to estimate the parameters

$$\beta_0, \beta_1, \beta_2, \beta_3$$

through GMM by solving the minimization problem of the GMM criterion function. We should use the identity matrix for the weights. We should treat each observation of $sick_i$ as a data moment. We should treat the predicted values as the model moments.

We should treat the error function of the moment be equal to the simple difference of the data moments from the model moments.

In [27]:
```python
# Import the necessary libraries
# Import packages and load the data
import numpy as np
import numpy.linalg as lin
import scipy.stats as sts
import scipy.integrate as intgr
import scipy.optimize as opt

import requests

# Download and save the data file incomes.txt
url = ('https://raw.githubusercontent.com/dtracht/persp-model-econ_W19/' +
       'master/ProblemSets/PS5/data/sick.txt')
data_file = requests.get(url, allow_redirects=True)
open('data/sick.txt', 'wb').write(data_file.content)

# Load the data as a NumPy array
pts = np.genfromtxt("data/sick.txt", delimiter=",", skip_header=1)
```

In [28]:
```python
# takes matrix of data
# returns first column as data moments
def data_moments(points):
    y_obs = points[:,0]
    return y_obs

# takes matrix of data, and four beta parameters
# returns linear combination
def model_moments(points, b0, b1, b2, b3):
    y_hats = b0 + b1*points[:,1] + b2*points[:,2] + b3*points[:,3]
    return y_hats

# takes matrix of data and four beta parameters
# returns error vector of residuals
def err_vec(points, b0, b1, b2, b3):
    y_obs = data_moments(points)
    y_hats = model_moments(points, b0, b1, b2, b3)
    err_vec = y_obs - y_hats
    return err_vec

# takes vector of four beta parameters
# takes tuple of matrix of data and weight matrix
# returns criterion value using error vector and weights
def criterion(params, *args):
    b0, b1, b2, b3 = params
    points, W = args
    err = err_vec(points, b0, b1, b2, b3)
    crit_val = err.T @ W @ err
    return crit_val
```

```python
In [29]: # intitlize at 0
         b0_init = 0
         b1_init = 0
         b2_init = 0
         b3_init = 0
         params_init = np.array([b0_init, b1_init, b2_init, b3_init])
         # use identity matrix for weights
         W_hat1 = np.eye(len(pts[:,0]))
         gmm_args = (pts, W_hat1)
         results = opt.minimize(criterion, params_init, args=(gmm_args), tol=1e-14)
         b0_GMM, b1_GMM, b2_GMM, b3_GMM= results.x
```

We wish examine the estimates for the parameters and the value of the GMM criterion function:

```python
In [30]: err = err_vec(pts, b0_GMM, b1_GMM, b2_GMM, b3_GMM)
         crit_val = err.T @ W_hat1 @ err
         print("beta_0=", b0_GMM)
         print("beta_1=", b1_GMM)
         print("beta_2=", b2_GMM)
         print("beta_3=", b3_GMM)
         print("Criterion value as estimated parameters=", crit_val)
```

```
beta_0= 0.25164430275796285
beta_1= 0.012933582043585837
beta_2= 0.400499865819143
beta_3= -0.009991760073144174
Criterion value as estimated parameters= 0.001821289680072053
```