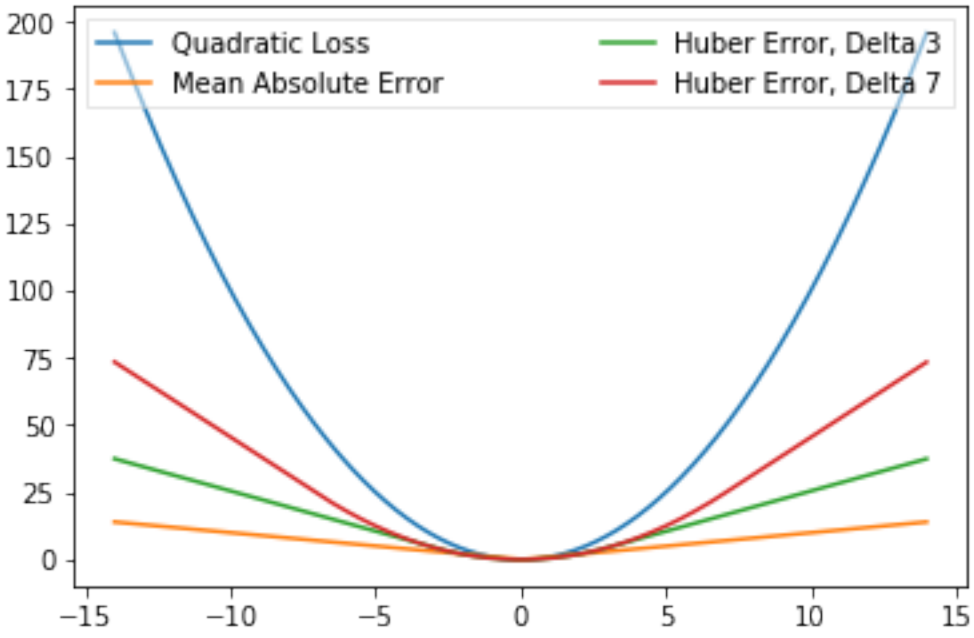
In [4]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import random
```

4a

In [213]:

```
delta = 3
huber = lambda e, delta: ((e**2)/2) if np.abs(e) <= delta else ((delta * np.abs(
e)) - ((delta**2)/2))


ax = plt.subplot(111)
t1 = np.arange(-14.0, 14.0, 0.01)


t2 = [huber(x, delta) for x in t1]

plt.plot(t1, t1**2, label="Quadratic Loss")
plt.plot(t1, np.abs(t1), label="Mean Absolute Error")
plt.plot(t1, [huber(x, 3) for x in t1], label="Huber Error, Delta 3")
plt.plot(t1, [huber(x, 7) for x in t1], label="Huber Error, Delta 7")

leg = plt.legend(loc='best', ncol=2, mode="expand", shadow=False, fancybox=False
)
leg.get_frame().set_alpha(0.5)


plt.show()
# plot second time
```



Relative advantages and disadvantages of loss functions for linear regression.

Given the plots and thier mathematical equations, it appears that Mean Absolute Error is less senstive to outliers than Quadratic Loss. A square of a single large outlier might be enough to alter our weights in a significant way. Therefore in noisy datasets with alrge outliers, it appears that MAE is a better choice. However, since mean absolute error doesn't reduce the slope as it approaches optima, it might be more sentive to large learning rates and might never reach an optima.

Huber Loss on the other hand is a good tradeoff between the above two loss functions. It is linear at large loss values allowing it to be robust to outliers, while employing a nice curvature with a reducing slope as it approaches optima. The catch with huber loss, however, is that we may not neccesarily know the delta beforehand.

4b

In [2]:

```
def genData(numPoints, slope, intercept):
    x = np.zeros(shape=(numPoints, 2))
    y = np.zeros(shape=numPoints)

    x[:,0] = 1
    x[:,1] = np.random.uniform(-2, 2, size = numPoints)

    for i in range(0, numPoints): y[i] = (x[i][1] * slope) + (intercept + np.ran
dom.normal(0,4))

    return x, y
```

In [207]:

```
def gradientDescent(x, y, theta, alpha, m, numIterations, loss_type):

    xTrans = x.transpose()
    for i in range(0, numIterations):

        hypothesis = np.dot(x, theta)

        if loss_type == 'quadratic':

            loss = hypothesis - y
            cost = np.sum(loss ** 2) / (2 * m)

        elif loss_type == 'mean_absolute':

            loss = hypothesis - y

            loss[loss>0] = 1
            loss[loss<0] = -1

            cost = np.sum(loss) / (2 * m)

        elif loss_type == 'huber':

            delta = 2
            huber = lambda e, delta: ((e/2)**2) if np.abs(e) <= delta else ((del
ta * np.abs(e)) - ((delta**2)/2))
            loss = hypothesis - y
            loss[abs(loss) <= delta] = loss[abs(loss) <= delta]
            loss[abs(loss) > delta] = delta * np.sign(loss[abs(loss) > delta])
            cost = huber(np.sum(loss), delta) / (2 * m)

        #print("\n\n\nIteration %d | Cost: %f" % (i, cost))

        gradient = np.dot(xTrans, loss) / m #Partial derivative
        theta = theta - alpha * gradient

        #print ('gradient' + str(gradient))
        #print ('theta' + str(theta))

    return theta
```

5A

In [221]:

```
#Referenced Stackoverflow for this

from numpy import array, dot, transpose
from numpy.linalg import inv

def analytical_linear_regression(x_train, y_train):

    X = np.array(x_train)
    y = np.array(y_train)

    Xt = transpose(X)
    product = dot(Xt, X)
    theInverse = inv(product)
    w = dot(dot(theInverse, Xt), y)

    return w
```

In [222]:

```
analytical_linear_regression(x,y)
```

Out[222]:

```
array([3.23055664, 4.20992994])
```

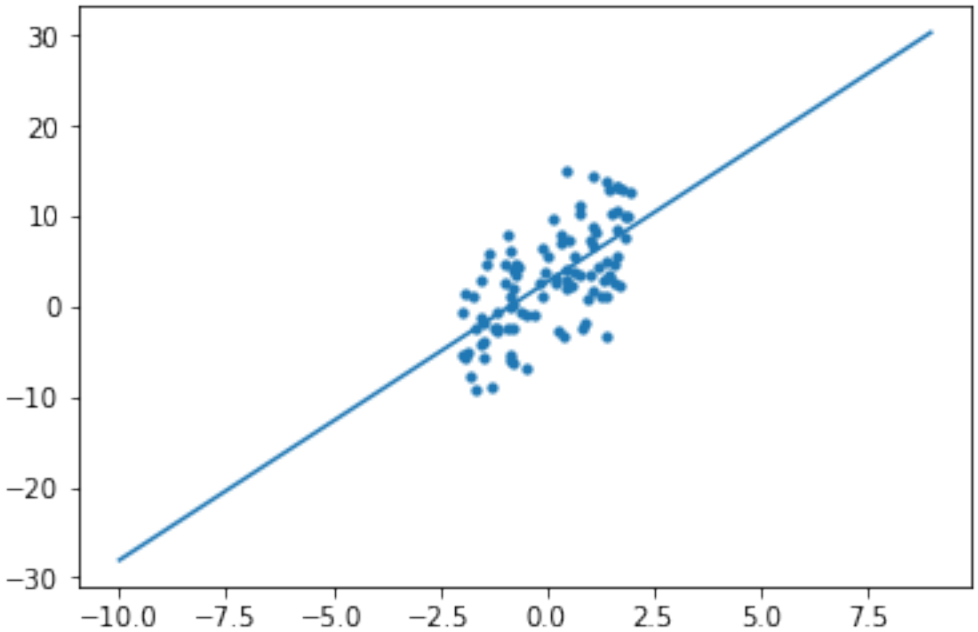
In [203]:

```
def gradientDescentSto(x, y, theta, alpha, m, numIterations, loss_type, delta=2,
conv=0.00000001):


    xTrans = x.transpose()
    count = 0
    theta_prev = theta + delta + 1

    while (count < numIterations):

            count += 1

            for i in range(m):

                hypothesis = x[i][0] * theta[0] + x[i][1]*theta[1]

                if loss_type == 'quadratic': loss = hypothesis - y[i]

                if loss_type == 'mean_absolute':

                    loss = hypothesis - y[i]
                    loss = 1 if loss >= 0 else -1

                if loss_type == 'huber':

                    loss = hypothesis - y[i]
                    if loss < -delta: loss = -delta
                    elif -delta <= loss and loss <= delta: loss = loss
                    elif delta < loss: loss = delta

                gradient1 = x[i][0] * loss
                gradient0 = x[i][1] * loss

                theta_prev = theta
                theta[1] = theta[1] - alpha * gradient0
                theta[0] = theta[0] - alpha * gradient1


    return theta
```

Calling and Plotting

In [206]:

```
x, y = genData(100, 3, 2)
m, n = np.shape(x)

theta = gradientDescentSto(x, y, np.ones(n), alpha = 0.0001, m, numIterations =
5000, 'huber', 7)
hypothesis = lambda x: x * theta[1] + theta[0]
plt.plot([i for i in range(-10,10)],[hypothesis(i) for i in range(-10,10)])
plt.scatter([i[1] for i in x], y, s=10)
plt.show()
```



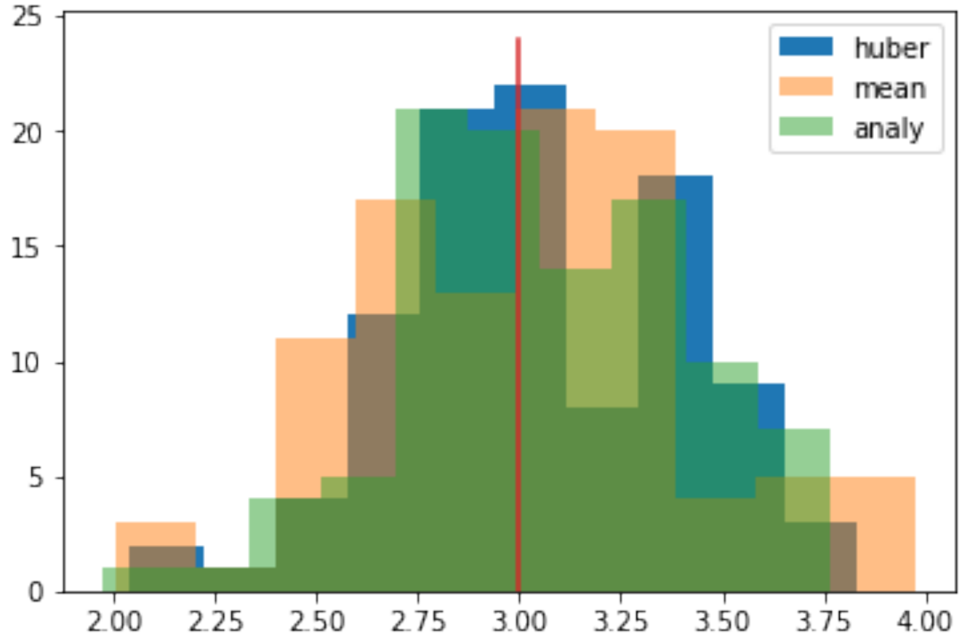
Call 1000 times and histogram for Huber, MAE and Analytical

In [492]:

```
thetas_huber = []
thetas_mean_abs = []
thetas_analytical = []

for i in range(100):
    #print (i)
    x, y = genData(100, 3, 2)
    m, n = np.shape(x)
    numIterations= 5000

    alpha = 0.0001
    thetas_huber.append(gradientDescentSto(x, y, np.ones(n), alpha, m, numIterat
ions, 'huber', 7))
    thetas_mean_abs.append(gradientDescentSto(x, y, np.ones(n), alpha, m, numIte
rations, 'mean_absolute'))
    thetas_analytical.append(analytical_linear_regression(x,y))
```

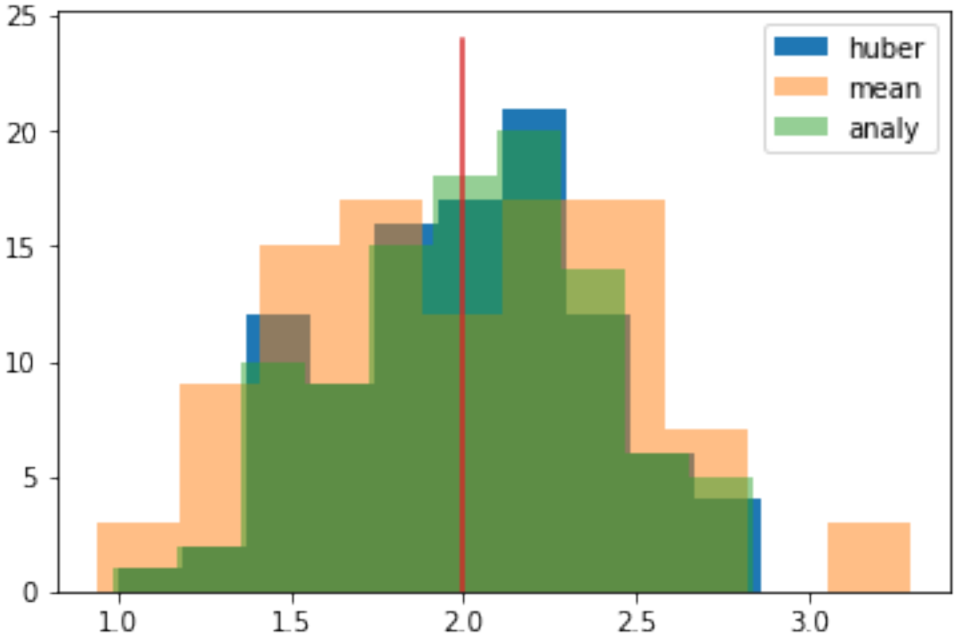
In [411]:

```
plt.hist(np.array(thetas_huber)[:,1], label = 'huber')
plt.hist(np.array(thetas_mean_abs)[:,1], label = 'mean', alpha = 0.5)
plt.hist(np.array(thetas_analytical)[:,1], label = 'analy', alpha = 0.5)

plt.plot([3 for i in range (25)], [i for i in range(25)])
#plt.plot(x,3,linestyle='solid')

plt.legend()
plt.show()
```



In [412]:

```
plt.hist(np.array(thetas_huber)[:,0], label = 'huber')
plt.hist(np.array(thetas_mean_abs)[:,0], label = 'mean', alpha = 0.5)
plt.hist(np.array(thetas_analytical)[:,0], label = 'analy', alpha = 0.5)


plt.plot([2 for i in range (25)], [i for i in range(25)])


plt.legend()
plt.show()
```



Choice of Loss Function Affecting Slope and Intercept

As explained in 4a, Hubers loss function appears to have the highest frequecy of slope closest to 3 and itercept with analytical coming in a close second. However, since analytical solution is the actual solution to our simulated values, that should be used as benchmark. As such, Huber outperforms MAE in estimating the slope and intercept.

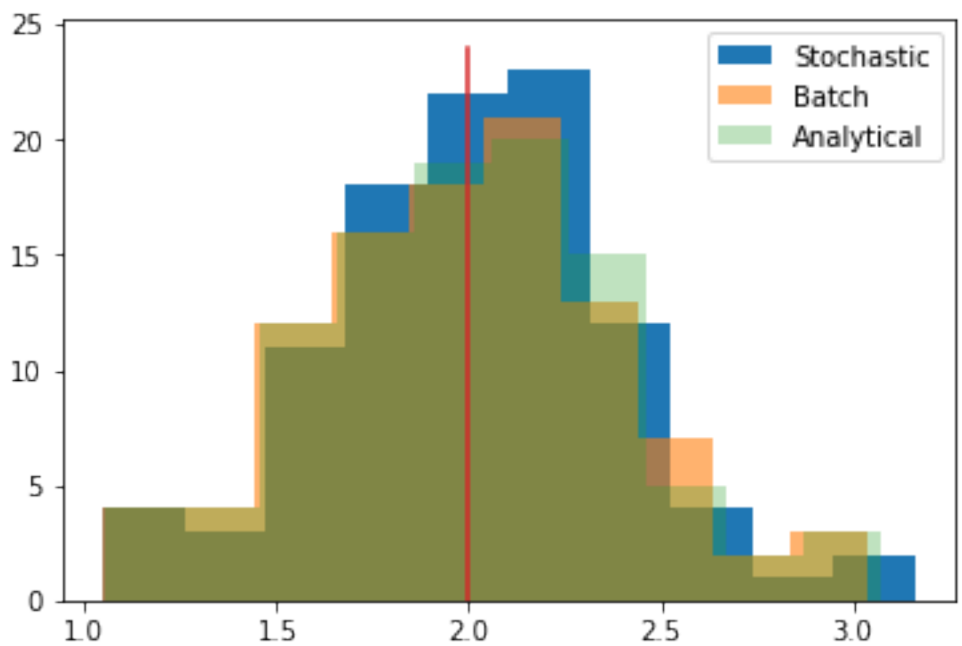
Call 1000 times and histogram for Batch, Stochastic and Analytical

In [228]:

```
thetas_sto_quadratic = []
thetas_batch_quadratic = []
thetas_analytical = []

for i in range(100):
    #print (i)
    x, y = genData(100, 3, 2)
    m, n = np.shape(x)
    numIterations= 5000
    alpha = 0.001

    thetas_sto_quadratic.append(gradientDescentSto(x, y, np.ones(n), alpha, m, n
umIterations, 'quadratic'))
    thetas_batch_quadratic.append(gradientDescent(x, y, np.ones(n), alpha, m, nu
mIterations, 'quadratic'))
    thetas_analytical.append(analytical_linear_regression(x,y))
```

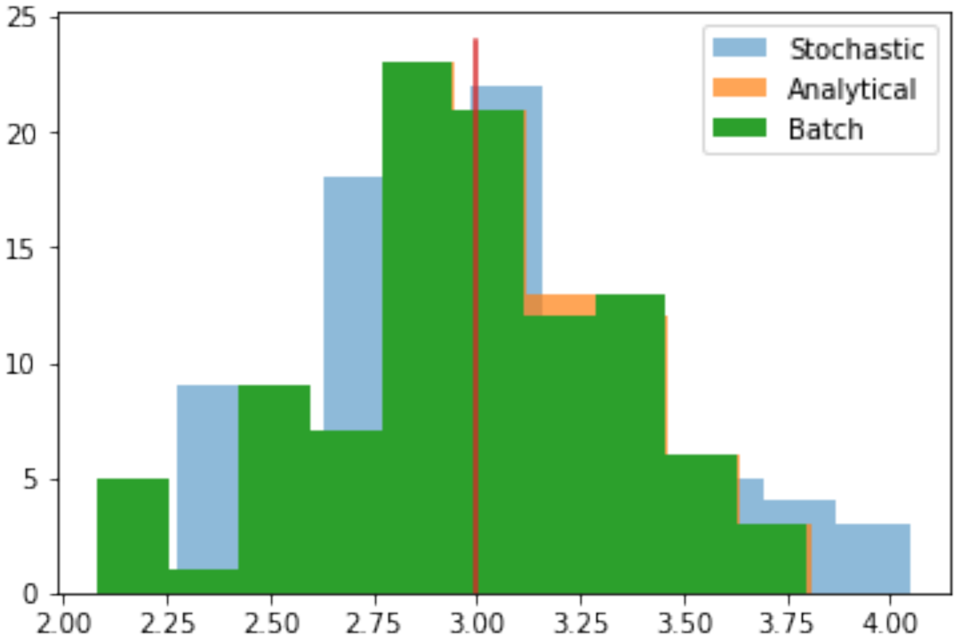
In [233]:

```
plt.hist(np.array(thetas_quadratic)[:,0], label = 'Stochastic', alpha = 1)
plt.hist(np.array(thetas_batch_quadratic)[:,0], label = 'Batch', alpha = 0.6)
plt.hist(np.array(thetas_analytical)[:,0], label = 'Analytical', alpha = 0.3)



plt.plot([2 for i in range (25)], [i for i in range(25)])


plt.legend()
plt.show()
```



In [242]:

```
plt.hist(np.array(thetas_quadratic)[:,1], label = 'Stochastic', alpha = 0.5)
plt.hist(np.array(thetas_analytical)[:,1], label = 'Analytical', alpha = 0.7)
plt.hist(np.array(thetas_batch_quadratic)[:,1], label = 'Batch', alpha = 1)



plt.plot([3 for i in range (25)], [i for i in range(25)])


plt.legend()
plt.show()
```



Choice of Algorithm Affecting Slope and Intercept

Using the analytical solution as the benchmark, it appears like batch almost exactly mimics the analytical solutions to the simulated data, outperforming stochastic in this case.

Add Anomalies

In [486]:

```
outlier_indices = [int(i) for i in np.random.uniform(0,100, size = 10)]
outlier_indices_indices = [int(i) for i in np.random.uniform(0,10, size = 5)]
x[outlier_indices]
```

Out[486]:

```
array([[ 1.        ,  1.40664088],
       [ 1.        ,  1.81175379],
       [ 1.        , -0.26609268],
       [ 1.        ,  0.57753574],
       [ 1.        ,  0.41188323],
       [ 1.        ,  0.03637198],
       [ 1.        ,  1.19956399],
       [ 1.        , -0.83287292],
       [ 1.        ,  0.7396422 ],
       [ 1.        , -0.43608485]])
```

In [488]:

```
positive_indices = [outlier_indices[i] for i in outlier_indices_indices]
negative_indices = [outlier_indices[i] for i in [outlier_indices.index(i) for i
in outlier_indices if outlier_indices.index(i) not in outlier_indices_indices]]
```

In [489]:

```
for i in positive_indices: x[i] += 0.5*x[i]
for i in negative_indices: x[i] -= 0.5*x[i]
```

Rerun 1000 times with anomalies

In [243]:

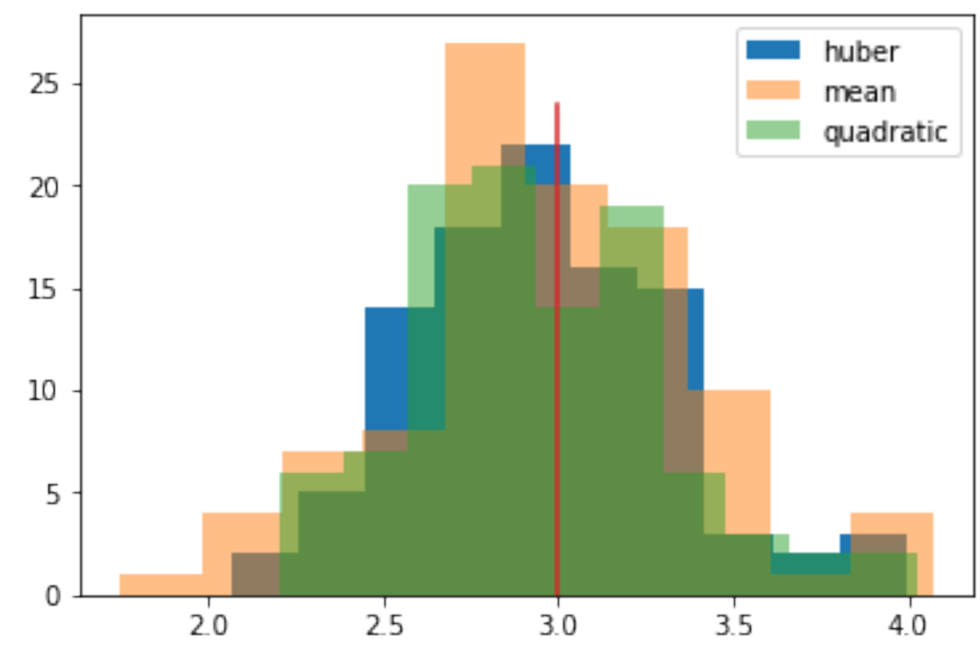
```
thetas_quadratic = []
thetas_quadratic.append(gradientDescentSto(x, y, np.ones(n), alpha, m, numIterat
ions, 'quadratic'))
```

In [245]:

```
thetas_huber = []
thetas_mean_abs = []
thetas_quadratic = []
thetas_analytical = []

for i in range(100):
    #print (i)
    x, y = genData(100, 3, 2)
    m, n = np.shape(x)
    numIterations= 5000
    alpha = 0.0001

    thetas_huber.append(gradientDescentSto(x, y, np.ones(n), alpha, m, numIterat
ions, 'huber', 7))
    thetas_mean_abs.append(gradientDescentSto(x, y, np.ones(n), alpha, m, numIte
rations, 'mean_absolute'))
    thetas_quadratic.append(gradientDescentSto(x, y, np.ones(n), alpha, m, numIt
erations, 'quadratic'))
    thetas_analytical.append(analytical_linear_regression(x,y))
```

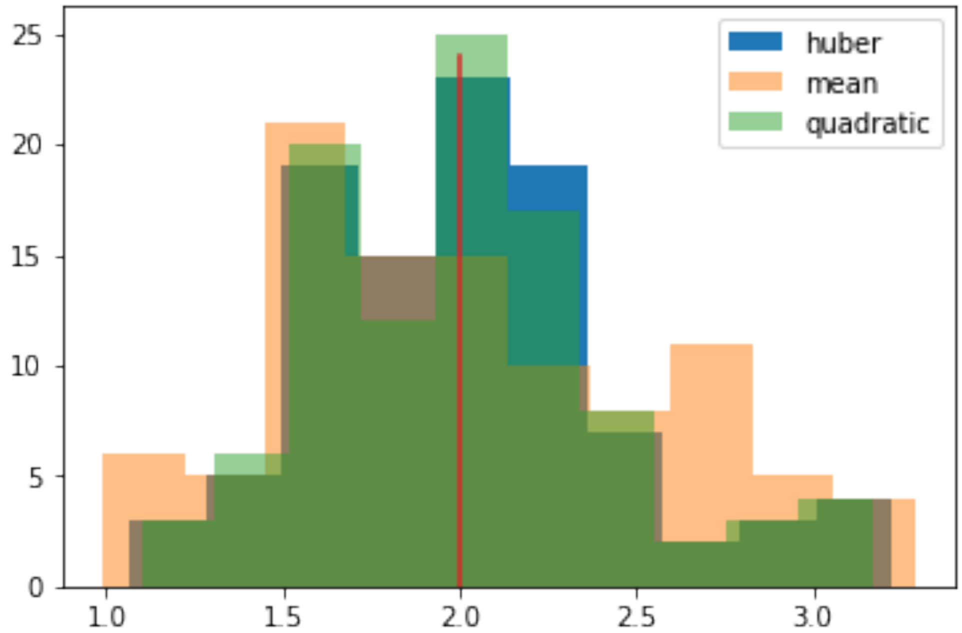
In [251]:

```
plt.hist(np.array(thetas_huber)[:,1], label = 'huber')
plt.hist(np.array(thetas_mean_abs)[:,1], label = 'mean', alpha = 0.5)
plt.hist(np.array(thetas_quadratic)[:,1], label = 'quadratic', alpha = 0.5)


plt.plot([3 for i in range (25)], [i for i in range(25)])


#plt.plot(x,3,linestyle='solid')
plt.legend()
plt.show()
```



In [250]:

```
plt.hist(np.array(thetas_huber)[:,0], label = 'huber')
plt.hist(np.array(thetas_mean_abs)[:,0], label = 'mean', alpha = 0.5)
plt.hist(np.array(thetas_quadratic)[:,0], label = 'quadratic', alpha = 0.5)


plt.plot([2 for i in range (25)], [i for i in range(25)])


plt.legend()
plt.show()
```



Choice of Loss in Noisy Data Affecting Slope and Intercept

Using the analytical solution as the benchmark, huber and quadratic outperform MAE in this case. This could be since the anomalies built into the data are balanced by design and therefore don't skew the squared error in any particular direction.

1

i). The conditional probability of Y_1 and Y_2 is

$$f_{Y_1|Y_2}(y_1|y_2) = \frac{1}{\sqrt{2\pi(1-\rho^2)}\,\sigma_1} e^{\left(\frac{-(y_1-a)^2}{2\sigma^2(1-\rho^2)}\right)}$$

where $a = \mu_1 + \rho\frac{\sigma_1}{\sigma_2}(y_2-\mu_2)$.

The expected value of $Y_1|Y_2$ is

$$E[Y_1|Y_2] = \mu_1 + \frac{\rho\sigma_1}{\sigma_2}(y_2-\mu_2)$$

$$E[Y_1|Y_2] = \left[\mu_1 - \mu_2\rho\frac{\sigma_1}{\sigma_2}\right] + \rho\frac{\sigma_1}{\sigma_2}y_2$$

This is a linear combination of Y_2, which mimics linear regression with one predictor. The variance is defined by $\sigma[Y_1|Y_2] = \sigma^2(1-\rho^2)$

1

a) Joint probability distribution of Y_1 and Y_2 is

$$f_{Y_1,Y_2}(y_1, y_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \cdot e^{\frac{-1}{2(1-\rho^2)}}$$

$$\left[\left(\frac{y_1-\mu_1}{\sigma_1}\right)^2 + \left(\frac{y_2-\mu_2}{\sigma_2}\right)^2 - 2\rho\left(\frac{y_1-\mu_1}{\sigma_1}\right)\left(\frac{y_2-\mu_2}{\sigma_2}\right)\right]$$

b) ~~[illegible]~~

$$f_{Y_1|Y_2}(y_1, y_2) = \frac{f_{Y_1,Y_2}(y_1, y_2)}{f_{Y_2}(y_2)}$$

$f_{Y_2}(y_2)$ (Marginal distribution) is.

$$\int_{-\infty}^{\infty} f_{Y_1,Y_2}(y_1, y_2)\, dy_1 = \frac{1}{\sqrt{2\pi}\sigma_2} e^{-\frac{(y_2-\mu_2)^2}{2\sigma_2^2}}$$

Applying Bayes theorum we get.

$$f_{Y_1 Y_2}(y_1, y_2) = \frac{\dfrac{1}{\sqrt{2}\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \cdot e^{\frac{-1}{2(1-\rho^2)}\left[\left(\frac{y_1-\mu_1}{\sigma_1}\right)^2 - 2\rho\left(\frac{y_1-\mu_1}{\sigma_1}\right)\left(\frac{y_2-\mu_2}{\sigma_2}\right) + \left(\frac{y_2-\mu_2}{\sigma_2}\right)^2\right]}}{\dfrac{1}{\sqrt{2\pi}\sigma_2} e^{-\frac{(y_2-\mu_2)^2}{2\sigma_2^2}}}$$

$$= \frac{1}{\sqrt{2\pi(1-\rho^2)}\sigma_1} e^{-\frac{(y_1-(\mu_1+\frac{\sigma_1}{\sigma_2}(y_2-\mu_2)))^2}{2\sigma_1^2(1-\rho^2)}}$$

The conditional probability ~~is~~ of $Y_1 | Y_2$ is normally distributed having mean $\mu_1 + \rho\frac{\sigma_1}{\sigma_2}(y_2-\mu_2)$ and $\sigma^2 = (1-\rho^2)\sigma_1^2$

Q.
2

a) Quadratic loss is defined by.

$$L = \sum_{i=1}^{N} e_i^2 .$$

Taking second derivative.

$$\frac{\partial L}{\partial e} = 2e \; ; \quad \frac{\partial^2 L}{\partial e^2} = 2$$

Since the $\frac{\partial^2 L}{\partial e^2} \geq 0$ this loss is convex.

b) L_1 norm $L = \sum_{i=1}^{N} |e_i|$

$$\frac{\partial L}{\partial e} = \text{sgn}(e) \; ; \quad \frac{\partial^2 L}{\partial e^2} = 0$$

Since $\frac{\partial^2 L}{\partial e^2} = 0$ this loss function is

also convex.

c) $L = \sum_{i=1}^{N} \ell(e_i)$ where $\ell(e) = \begin{cases} \frac{1}{2}e^2 & \text{if } |e| \leq \delta \\ \delta|e| - \frac{1}{2}\delta^2, & \text{if } |e| > \delta \end{cases}$

$$\frac{\partial L}{\partial e} = \begin{cases} e & , \text{if } |e| \leq \delta \\ \delta \, \mathrm{sgn}(e), & \text{if } |e| > \delta \end{cases}$$

$$\frac{\partial^2 L}{\partial e^2} \begin{cases} 1 ; & \text{if } |e| \leq \delta \\ 0 & \text{if } |e| > \delta \end{cases}$$, both of which are greater than or equal to 0.

Therefore this function is convex as well.

3

a). $y_i = \theta_0 + \theta_1 x_i + e_i \quad , i = 1 \ldots N.$

Our goal is to minimise the cost function

$$J(\theta_0 \ldots n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)} - y^{(i)})^2.$$

where m is the number of datapoints

Our hypothesis $h(x) = \theta_0 x_0 + \theta_1 x_1 \ldots \theta_n x_n$

$$h(x) = \theta^T X.$$

where $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$ and $X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

The cost function converted into matrix notation, we get.

$$J(\theta) = \frac{1}{2m} (X\theta - y)^T (X\theta - y)$$

$$J(\theta) = \frac{(X\theta)^T - y^T (X\theta - y)}{2m}$$

$$= \frac{(X\theta)^T X\theta - (X\theta)^T y - y^T (X\theta) + y^T y}{2m}$$

$$= \frac{\theta^T X^T X\theta - 2(X\theta)^T y + y^T y}{2m}$$

Differentiating and setting it to 0

$$= \frac{(2X^T X\theta - 2X^T y)}{2m} = 0$$

$$2X^T X\theta - 2X^T y = 0$$

$$\theta = (X^T X)^{-1} X^T y$$

b) Gradient descent is used to find optimal θ's in a given linear equation of form

$$h(x) = \theta_n x_n + \dots + \theta_2 x_2 + \theta_1 x_1 + \theta_0 x_0$$

such that the cost function $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} h_\theta(x^i - y^i)^2$ is minimised

In order to minimise the above cost function, we need to calculate the slope of the function and adjust θ ~~m~~ ~~[illegible]~~ such that we 'move down' the cost slope.

This is done by differentiating the cost function, which gives us

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = \frac{1}{m} \sum_{i=0}^{m} (h_\theta(x^i) - y^i) \cdot x^i$$

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta x^i - y^i)$$

The above slopees are multipled by a set learning rate α and subtracted by θ_0 and θ_1

This gives us

$$\theta_0 = \theta_0 - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right)$$

$$\theta_1 = \theta_1 - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^i) - y^i \right) . x^i$$

Both θ_0 and θ_1 are updated simultaneously until convergence.

c) Stochastic gradient descent is, for the most part, the same as batch gradient descent.

The major difference is in the way stochastic processes its iterations. Whereas batch gradient descent processes all datapoints at once, for each iteration, stochastic processes one row per iteration.

Formally its specified as follows.

while $|\text{new } \theta - \text{old } \theta| > \text{convergence}$:

{

for $i=0 : i<n ; i{+}{+}$.

{ $\theta_{j=0 \text{ to } m} = \theta_j - \alpha \left(h_\theta(x^i) - (y^i) \right) x_j^i$

}

}