HW08- CS 189

1.
Who else did you work with on this homework? In case of course events, just describe the group. How did you work on this homework? Any comments about the homework?

I worked on this homework with Ehimare Okoyomon, Prashanth Ganeth, and Daniel Mockaitis. We worked by getting together throughout the week and communicating on facebook.

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up}

Nicholas Lorio, 26089160

Question 5 Own Question.

Compare batch gradient descent and stochastic gradient descent. Why should we use SGD?

Batch gradient descent finds the gradient using either the whole data set or a minibatch of the full data set. It works well for convex, relatively smooth functions. Given an appropriate learning rate it will eventually find the minimum.

SGD however computes gradient using only a single sample. It is in most cases a computationally faster approach due to the minimize number of computations per step. Another principle attraction of this method is its large stochastic noise. For functions with lots of local maxima/minima the batch gradient descent method will often times get stuck at these points as the batch size tends to average out the noise per step. The noise, jerking property of single sample gradient descent, lets the algorithm jerk out of local minmi into a region which is hopefully more optimal.

The best approach is to go for a mini batch gradient descent that is small enough to avoid poor local minima but large enough to ensure it stays in the global solution.

In essence we can perform many more iterations of SGD than of regular or batch GD due to its computational simplicity in order to find the optima.

N. Reid, 20089160

HW08 - 189

2a. Show $\|w^{t+1} - w^*\|_2^2 = \|w^t - w^*\|_2^2 - 2\alpha_t \langle G_t, w^t - w^* \rangle + \alpha_t^2 \|G_t\|_2^2$

① ②

$w^{t+1} = w^t - \alpha_t G_t$

$\|w^{t+1} - w^*\|_2^2 = \|w^t - \alpha_t G_t - w^*\|_2^2 = (w^t - \alpha_t G_t - w^*)^T (w^t - \alpha_t G_t - w^*)$

$= (w^{tT} w^t - w^{tT} \alpha_t G_t - w^{tT} w^* - G_t^T \alpha_t^T w^t + G_t^T \alpha_t^T \alpha_t G_t + G_t^T \alpha_t^T w^* - w^{*T} w^t + w^{*T} \alpha_t G_t + w^{*T} w^*)$

$= \|w^t - w^*\|_2^2 - w^{tT} \alpha_t G_t - G_t^T \alpha_t^T w^t + G_t^T \alpha_t^T \alpha_t G_t + G_t^T \alpha_t^T w^* + w^{*T} \alpha_t G_t$

$= \|w^t - w^*\|_2^2 + \|\alpha G_t\|_2^2 - 2\alpha_t G_t^T w^t + 2\alpha_t G_t^T w^*$

$= \|w^t - w^*\|_2^2 + \alpha_t^2 \|G_t\|_2^2 - 2\alpha_t \langle G_t, w^t - w^* \rangle$

$\therefore$ ① = ② as required.

G_t is one sample of r.v. $G(w_t)$

e.g. part I. $E_{i_t} \nabla F_{i_t}(w) = \nabla F(w)$ ← unbiased SGD by def =

$$E[\langle G(w^t), w^t - w^* \rangle] = E_{i_1 \ldots i_{t-1}}[E_{i_t}[\langle G(w^t), w^t - w^* \rangle | i_1, \ldots, i_{t-1}]]$$

$$= P(i_1)[E_t \langle G_1, w^1 - w^* \rangle | i_1] + \ldots + P(i_t)[E_{i_t}[\langle G_{i_t}, w^{t-1} - w^* \rangle$$

$$= \sum_{t=1}^{t-1} \frac{1}{n} [E_t[\langle G_t, w^t - w^* \rangle | i_t]] \quad \text{as } E[X] = E[E[X|Y]] = \sum_i E[X|A_i] P(A_i)$$

i_t uniformly sampled from all samples $1, \ldots, n$ & is (indep.) drawn from each iteration t.

$$= \sum_{t=1}^{t-1} \frac{1}{n} \cdot E_t[\langle \nabla F_{i_t}, w^t - w^* \rangle | i_t] = \frac{1}{n}$$

$$= \sum_{t=1}^{t-1} \langle \nabla F(w^t), w^t - w^* \rangle \frac{1}{n} = E[\langle \nabla F(w^t), w^t - w^* \rangle]$$

as required.

alternate proof

2.b. part i) $\Delta_t = E\|w_t - w^*\|_2^2$

$E_{i_t}\langle G(w_t), w_t - w^* \rangle = E_{i_t}[G(w_t)^T (w_t - w^*)]$

$= E_{i_t}[G(w_t)^T w_t - G(w_t)^T w^*] = E_{i_t}[G(w_t)^T w_t] - E[G(w_t)^T w^*]$

From Hint $E_{i_t}[\partial f_{i_t}(w)] = \partial F(w)$ & by def: $G(w_t) = \partial f_{i_t}(w_t)$

$\Rightarrow E_{i_t}[G(w_t)^T] w_t - E_{i_t}[G(w_t)^T] w^* = E[\partial f_{i_t}(w_t)^T] w_t - E[\partial f_{i_t}(w_t)^T] w^*$

$= \partial F(w)^T w_t - \partial F(w)^T w^* = \partial F(w)^T (w_t - w^*)$

$= E_{i_t}[\partial F(w_t)^T (w_t - w^*)] = E_{i_t}[\langle \partial F(w_t), w_t - w^* \rangle]$

2.5. Cont.

Proof $\Delta_{t+1} \leq (1+\alpha_t^2 M_g^2)\Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w_t), w^t - w^* \rangle] + \alpha_t^2 \beta^2$

$\Delta_t = \mathbb{E}[\|w^t - w^*\|_2^2]$

$\Delta_{t+1} = \mathbb{E}[\|w^{t+1} - w^*\|_2^2] = \mathbb{E}[\|w^t - w^*\|_2^2 - 2\alpha_t \langle G_t, w^t - w^* \rangle + \alpha_t^2 \|G_t\|_2^2]$

$= \mathbb{E}[\|w^t - w^*\|_2^2] - 2\alpha_t \mathbb{E}[\langle G_t, w^t - w^* \rangle] + \alpha_t^2 \mathbb{E}[\|G_t\|_2^2]$

// From eqn 2" / Assumption 2"

$\leq \Delta_t - 2\alpha_t \mathbb{E}[\langle G_t, w^t - w^* \rangle] + \alpha_t^2 (M_g^2 \|w^t - w^*\|_2^2 + \beta^2)$

// From 2.5.1"

$\leq \Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w^t), w^t - w^* \rangle] + \alpha_t^2 M_g^2 \|w^t - w^*\|_2^2 + \alpha_t^2 \beta^2$

$\mathbb{E}[\Delta_{t+1}] = \Delta_{t+1} \leq \mathbb{E}[\Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w^t), w^t - w^* \rangle] + \alpha_t^2 M_g^2 \|w^t - w^*\|_2^2 + \alpha_t^2 \beta^2]$

expectation w.r.t w (& scalars) = same

$\leq \Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w^t), w^t - w^* \rangle] + \alpha_t^2 M_g^2 \mathbb{E}[\|w^t - w^*\|_2^2] + \alpha_t^2 \beta^2$ (Δ_t)

$\Delta_{t+1} \leq (1+\alpha_t^2 M_g^2)\Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w^t), w^t - w^* \rangle] + \alpha_t^2 \beta^2$

as required.

2.c. Strong convexity with $\lambda_{min}(X^TX) > 0$ for OLS

Let $M := \lambda_{min}(X^TX) > 0$ (minimum eigenval X^TX)

$$\Delta_{t+2} \leq (1 + \alpha_t^2 M_g^2)\Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w_t), w_t - w^* \rangle] + \alpha_t^2 B^2$$

• Hint: $\nabla F(w^*) = 0$ ($= M$)

$$\Delta_{t+1} \leq (1 + \alpha_t^2 M_g^2)\Delta_t - 2\alpha_t \mathbb{E}[\langle \nabla F(w_t) - \nabla F(w^*), w_t - w^* \rangle] + \alpha_t^2 B^2$$

$$\mathbb{E}[\langle \nabla F(w_t) - \nabla F(w^*), w_t - w^* \rangle] =$$

$$\mathbb{E}[\langle \nabla F(w_t), w_t \rangle - \langle \nabla F(w_t), w^* \rangle - \langle \nabla F(w^*), w_t \rangle + \langle \nabla F(w^*), w^* \rangle]$$

$$\nabla F(w_t) = 2X^T(Xw_t - y) \cdot \frac{1}{n}$$

$$\frac{1}{n}\mathbb{E}[(2(X^TXw_t - X^Ty)^Tw_t - (2X^TXw_t - X^Ty)^Tw^* - (2X^TXw^* - X^Ty)^Tw_t + (2X^TXw^* - X^Ty)^Tw^*]$$

$$\frac{2}{n}\mathbb{E}[(X^TXw_t - X^Ty)^T(w_t - w^*) - (X^TXw^* - X^Ty)^T(w_t - w^*)]$$

$$\frac{2}{n}\mathbb{E}[(X^TXw_t - X^Ty - X^TXw^* + X^Ty)^T(w_t - w^*)]$$

$$\frac{2}{n}\mathbb{E}[(w_t - w^*)^T(X^TX)(w_t - w^*)]$$ • Rayleigh quant

2.c. cont

$$= \frac{2}{n} E\left[(w^t - w^*)^T X^T X (w^t - w^*) \right]$$

$$= \frac{2}{n} E\left[R_{X^TX}(w^t - w^*) \| w^t - w^* \|_2^2 \right]$$

- $R(w^t - w^*) = \dfrac{(w^t - w^*)^T X^T X (w^t - w^*)}{(w^t - w^*)^T (w^t - w^*)} \geq R(w^d - w^*)$
- let d be the last of $\{1, \ldots t-1\}$, & this is our smallest / minimum eigenvalue d.

$$\geq \lambda_d (X^T X) = \lambda_{min}(X^T X)$$

$$\geq \frac{2}{n} E\left[\lambda_{min}(X^T X) \| w^t - w^* \|_2^2 \right]$$

$$\geq \frac{2}{n} \lambda_{min}(X^T X) E\left[\| w^t - w^* \|_2^2 \right] = \frac{2}{n} \lambda_{min}(X^T X) \Delta t$$

- as required to show our inequality

$$\Delta_{t+1} \leq \left(1 + \alpha_t^2 M_G^2 - 2\alpha \left(\frac{2}{n} \lambda_{min}(X^T X) \right) \right) \Delta t + \alpha_t^2 B^2$$

- the inequality is satisfied as the logarithm sign makes our inequality sign / bound for M satisfy the inequality of inequality (3)

2.d. $f_i(w) = (x_i^T w - y_i)$ $\quad y_i = x_i^T w^*$

$G(w) = \nabla f_{ik}(w)$

thus

$$E\|G(w)\|_2^2 \le \max_i \|\nabla f_i(w)\|_2^2 = \max \|2x_i(x_i^T w - y_i)\|_2^2$$

$$= \max_i \|x_i(x_i^T w - y_i)\|_2^2 = \max_i \|x_i(x_i^T w - x_i^T w^*)\|_2^2$$

$$= \max_i \|x_i x_i^T (w - w^*)\|_2^2 \le \max_i \|x_i x_i^T\| \|w - w^*\|_2^2$$

$$\le \max_i \|\|x_i\|_2^2\|_2^2 \|w - w^*\|_2^2 \le \max_i \|x_i\|_2^4 \|w - w^*\|_2^2$$

• cauchy schwarz

w/ $M_G^2 = \max_i \|x_i\|_2^4$ & $\beta = 0$ the inequality we want to satisfy is

$$E\|G(w)\|_2^2 \le M_G^2 \|w - w^*\| + \sigma^2$$

So the inequality is shown to be true above.

2. d. Cont.

Show $\Delta_t \leq \left(1 - \frac{M^2}{M_g^2}\right)^t \Delta_0$

$\therefore$

$$\Delta_{t+1} \leq \left(1 - M^2/M_g^2\right) \Delta_t$$

$$\Delta_1 \leq \left(1 + \alpha^2 M_g^2 - 2\alpha M\right) \Delta_0 + 0$$

• find optimal learning α.

$$\delta\left(1 + \alpha^2 M_g^2 - 2\alpha M\right)/\delta\alpha = \left(2\alpha M_g^2 - 2M\right) = 0$$

$$2\alpha M_g^2 = 2M \qquad \alpha = M/M_g^2$$

$$\Delta_1 \leq \left(1 + \frac{M^2 M_g^2}{M_g^4} - 2\cdot\frac{M^2}{M_g^2}\right) \Delta_0$$

$$\Delta_1 \leq \left(1 - M^2/M_g^2\right) \Delta_0$$

By Induction:

$$\Delta_t \leq \left(1 - M^2/M_g^2\right)^t \Delta_0 \qquad \text{as required.}$$

$x_t = x^*$, $M_g \neq 0$, $\beta \neq 0$

i.e. $f(w^*) = \frac{1}{n}\|Xw^* - y\|_2^2 > 0$ $\quad \gamma = 1 + \alpha^2 M_g^2 - 2\alpha M$

α large enough s.t. $\gamma < 1$

use part c. prove $\Delta_t \leq \gamma^t \Delta_0 + \frac{\alpha\beta^2}{2M - \alpha M_g^2}$

Part c: $\Delta_{t+1} \leq (1 + \alpha^2 M_g^2 - 2\alpha M)\Delta_t + \alpha^2\beta^2$

$\Delta_{t+1} \leq \gamma \Delta_t + \alpha^2\beta^2$

$\Delta_1 \leq \gamma \Delta_0 + \alpha^2\beta^2$ $\qquad \Delta_2 \leq \gamma \Delta_1 + \alpha^2\beta^2$

$\Delta_2 \leq \gamma(\gamma \Delta_0 + \alpha^2\beta^2) + \alpha^2\beta^2$

$\Delta_2 \leq \gamma^2 \Delta_0 + \alpha^2\beta^2(1 + \gamma)$ $\quad \gamma^2\Delta_0 + \alpha^2\beta^2 + \alpha^2\beta^2 M_g^2 - 2\alpha^2\beta^2 M$

$\Delta_3 \leq \gamma \Delta_2 + \alpha^2\beta^2 \leq \gamma(\gamma^2 \Delta_0 + \alpha^2\beta^2(1+\gamma)) + \alpha^2\beta^2$

$\leq \gamma^3 \Delta_0 + \alpha^2\beta^2(1+\gamma)\gamma + \alpha^2\beta^2$

$\leq \gamma^3 \Delta_0 + \alpha^2\beta^2(1 + \gamma + \gamma^2)$

$(1+\gamma)^2$

$\Delta_t \leq \gamma^t \Delta_0 + \alpha^2\beta^2(1+\gamma)^{t+1}$ using hint 2 that $\gamma < 2$

$\Delta_t \leq \gamma^t \Delta_0 + \alpha^2\beta^2 \sum_{b=0}^{t-1} \gamma^b \leq \gamma^t \Delta_0 + \frac{\alpha^2\beta^2}{1 - (1 + \alpha^2 M_g^2 - 2\alpha M)}$

$\leq \gamma^t \Delta_0 + \frac{\alpha\beta^2}{-\alpha M_g^2 + 2M}$ thus $\Delta_t \leq \gamma^t \Delta_0 + \frac{\alpha\beta^2}{2M - \alpha M_g^2}$

as required

2.e. Cont. • Does gaurntee convgence $w^t \to w^*$ as $t \to \infty$?

$\gamma < 1$ $\quad \Delta_t \leq \gamma^t \Delta_0 + \dfrac{\alpha B^2}{2m - \alpha M_g^2}$, $F(w^*) = \frac{1}{n}\|Xw^* - y\|_2^2 >$ minimum loss > 0

$\Delta_t = E\|w^t - w^*\|_2^2$ "the avge squared error"

• α is assumed to be s.t. $0 < \gamma < 1$

• $\dfrac{\alpha B^2}{2m - \alpha M_g^2}$, $\underbrace{\alpha B^2}_{\text{const}} \cdot \dfrac{1}{2m - \alpha M_g^2}$, $M = \dfrac{2\lambda_{min}(X^TX)}{n}$ > 0 (n - constant, sample size)

$\lambda_{min}(X^TX) > 0$

$$\lim_{t\to\infty} \Delta_t \leq \lim_{t\to\infty} \gamma^t \Delta_0 + \underbrace{\frac{\alpha B^2}{2m - \alpha M_g^2}}$$

$\exists$ no t values in this expression

$$\lim_{t\to\infty} E\|w^t - w^*\|_2^2 \leq \lim_{t\to\infty} \gamma^t \Delta_0 + \text{constant}$$

$0 < \gamma$, $\gamma < 1$ $\therefore$ the avge squared error converges to $\dfrac{\alpha B^2}{2m - \alpha M_g^2}$

Prob Model

3.d. $y_i - z_y = w^T(x_i - z_x)$ x, y_i observed data

$y_{true} = w^T x_{true}$

$\left[y_i = w^T x_i - w^T z_x + z_y \right]$

$y_i - z_y$ is ~~true data~~ y_{true}

$x_i - z_x$ is x_{true}

$z_y \sim N(0,1)$

$z_{x,j} \sim N(0,1)$ for all j

$w^T z_x - z_y = w^T x_i - y$

Soeas $y_i - w^T x_i = z_y - \sum w_j^T z_{x,j}$

$\underbrace{\sum_j w_j z_{x,j} - z_y}_{N(0,1)\ N(0,1)} = w^T x_i - y_i$

- sum of independent gaussian is gaussian (known property of gaussians)
- $\sum w_j z_{x,j} \sim \sum w_j N(0,1)$

$\sim N(0, \sum w_j^2)$

$\sim N(0, \sum_j w_j^2 + 1) \sim y_i$

$$p(y_i | x_i) = \frac{1}{\sqrt{2\pi\left(\sum_j w_j^2 + 1\right)}} \exp\left(-\frac{1}{2} \frac{(w^T x_i - y_i)^2}{\sum w_j^2 + 1} \right)$$

$$\log(p(y_i | x_i)) = -\frac{1}{2}\log\left(2\pi\left(\sum_j w_j^2 + 1\right)\right) - \frac{(w^T x_i - y_i)^2}{2\left(\sum w_j^2 + 1\right)}$$

$$\sum_{i=1}^{\hat{}} \log(p_w(y_i | x_i)) = \underbrace{\frac{-n \cdot \log(2\pi)}{2}}_{\text{constant}} - \sum_{i=1}^{\hat{}} \frac{1}{2}\log\left(\sum w_j^2 + 1\right) - \sum_{i=1}^{\hat{}} \frac{(w^T x_i - y_i)^2}{2\left(\sum w_j^2 + 1\right)}$$

$$= C - \frac{n}{2}\log\left(\|w_j\|_2^2 + 1\right) - \frac{1}{2\left(\|w_j\|_2^2 + 1\right)} \cdot \sum_{i=1}^{\hat{}} (w^T x_i - y_i)^2$$

as required.

$\sum (-(y_i - w^T x_i))^2$

$\sum (y_i - w^T x_i)^2$ same

In [24]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Question 2

Part A: one solution

Assuming that I want to find the w that minimizes $\frac{1}{2n}||Xw - y||_2^2$. In this part, X is full rank, and $y \in range(X)$

In [25]:
```
X = np.random.normal(scale = 20, size=(100,10))
print(np.linalg.matrix_rank(X)) # confirm that  the matrix is full rank
# Theoretical optimal solution
w = np.random.normal(scale = 10, size = (10,1))
y = X.dot(w)
```

```
10
```

In [85]:
```
def sgd(X, y, w_actual, threshold, max_iterations, step_size, gd=False):
    if isinstance(step_size, float):
        step_size_func = lambda i: step_size

    else:
        step_size_func = step_size

    # run 10 gradient descent at the same time, for averaging purpose
    # w_guesses stands for the current iterates (for each run)
    w_guesses = [np.zeros((X.shape[1], 1)) for _ in range(10)]
    n = X.shape[0]
    error = []
    it = 0
    above_threshold = True
    previous_w = np.array(w_guesses)

    while it < max_iterations and above_threshold:
        it += 1
        curr_error = 0
        for j in range(len(w_guesses)):
            if gd:
                # Your code, implement the gradient for GD
                sample_gradient =  2/n*(X.T.dot(X.dot(w_guesses[j])) - X.T.dot(y))

            else:
                # Your code, implement the gradient for SGD
                # Batch size of 1
                index = np.random.randint(0,n)
                X_sgd = X[index].reshape((10,1))
                y_sgd = y[index]

                sample_gradient = 2*X_sgd*(X_sgd.T.dot(w_guesses[j]) - y_sgd)
                #sample_gradient =  2 * (X_sgd.T @ previous_w[j] - y_sgd) * X_sgd
            # Your code: implement the gradient update

            # learning rate at this step is given by step_size_func(it)
            # w_guesses[j] = ?
            w_guesses[j] = previous_w[j] - step_size_func(it)*sample_gradient

            curr_error += np.linalg.norm(w_guesses[j]-w_actual)
        error.append(curr_error/10)

        diff = np.array(previous_w) - np.array(w_guesses)
        diff = np.mean(np.linalg.norm(diff, axis=1))
        above_threshold = (diff > threshold)
        previous_w = np.array(w_guesses)
    return w_guesses, error
```

In [86]:
```
its = 5000
w_guesses, error = sgd(X, y, w, 1e-10, its, 0.0001)
```

In [87]:
```
iterations = [i for i in range(len(error))]
#plt.semilogy(iterations, error, label = "Average error in w")
plt.semilogy(iterations, error, label = "Average error in w")
plt.xlabel("Iterations")
plt.ylabel("Norm of $w^t - w^*$",  usetex=False)
plt.title("Average Error vs Iterations for SGD with exact sol")
plt.legend()
plt.show()
```

In [88]:
```
print("Required iterations: ", len(error))
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses])
print("Final average error: ", average_error)
```

```
Required iterations:  596
Final average error:  1.0221264473109138e-09
```

Part B: No solutions, constant step size

In [89]:
```
y2 = y + np.random.normal(scale=5, size = y.shape)
w=np.linalg.inv(X.T @ X) @ X.T @ y2
```

In [90]:
```
its = 5000
w_guesses2, error2 = sgd(X, y2, w, 1e-5, its, 0.0001)
w_guesses3, error3 = sgd(X, y2, w, 1e-5, its, 0.00001)
w_guesses4, error4 = sgd(X, y2, w, 1e-5, its, 0.000001)
```

In [91]:
```
w_guess_gd, error_gd = sgd(X, y2, w, 1e-5, its, 0.001, True)
```

In [92]:
```
plt.semilogy([i for i in range(len(error2))], error2, label="SGD, lr = 0.0001")
plt.semilogy([i for i in range(len(error3))], error3, label="SGD, lr = 0.00001")
plt.semilogy([i for i in range(len(error4))], error4, label="SGD, lr = 0.000001")
plt.semilogy([i for i in range(len(error_gd))], error_gd, label="GD, lr = 0.00001")
plt.xlabel("Iterations")
plt.ylabel("Norm of $w^t - w^*$",  usetex=False)
plt.title("Total Error vs Iterations for SGD without exact sol")
plt.legend()
plt.show()
```

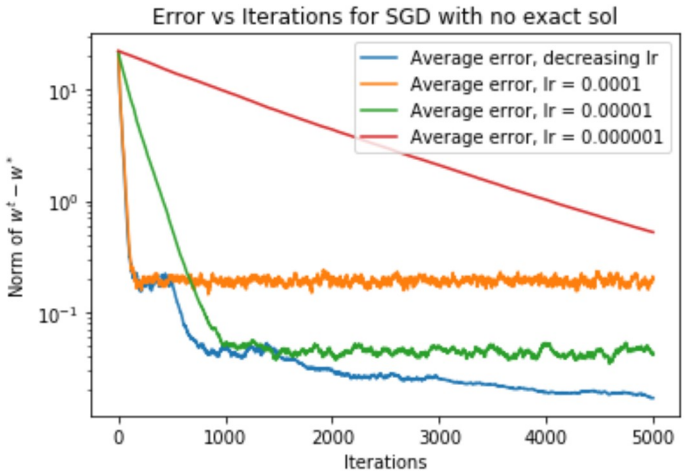
In [93]:
```
print("Required iterations, lr = 0.0001: ", len(error2))
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses2])
print("Final average error: ", average_error)

print("Required iterations, lr = 0.00001: ", len(error3))
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses3])
print("Final average error: ", average_error)

print("Required iterations, lr = 0.000001: ", len(error4))
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses4])
print("Final average error: ", average_error)

print("Required iterations, GD: ", len(error_gd))
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guess_gd])
print("Final average error: ", average_error)
```

```
Required iterations, lr = 0.0001:  5000
Final average error:  0.1979710675397775
Required iterations, lr = 0.00001:  5000
Final average error:  0.04161109811756734
Required iterations, lr = 0.000001:  5000
Final average error:  0.5226337598000734
Required iterations, GD:  18
Final average error:  9.441941045229165e-06
```

Part C: No solutions, decreasing step size

In [94]:
```
its = 5000
def step_size(step):
    if step < 500:
        return 1e-4
    if step < 1500:
        return 1e-5
    if step < 3000:
        return 3e-6
    return 1e-6

w_guesses_variable, error_variable = sgd(X, y2, w, 1e-10, its, step_size, False)
```

In [95]:
```
plt.semilogy([i for i in range(len(error_variable))], error_variable, label="Averag
e error, decreasing lr")
plt.semilogy([i for i in range(len(error2))], error2, label="Average error, lr = 0.
0001")
plt.semilogy([i for i in range(len(error3))], error3, label="Average error, lr = 0.
00001")
plt.semilogy([i for i in range(len(error4))], error4, label="Average error, lr = 0.
000001")

plt.xlabel("Iterations")
plt.ylabel("Norm of $w^t - w^*$",  usetex=False)
plt.title("Error vs Iterations for SGD with no exact sol")
plt.legend()
plt.show()
```



In [96]:
```
print("Required iterations, variable lr: ", len(error_variable))
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses_variabl
e])
print("Average error with decreasing lr:", average_error)
```

```
Required iterations, variable lr:  5000
Average error with decreasing lr: 0.016877360019704468
```

2: Conclusions

Part A

SGD error is plotted against the number of iterations of SGD completed. SGD with this learning rate has stochastsic properties and reaches the threshold of accuracy before the maximum number of iterations is reached.

Part B

No exact solution. There is no solution so GD does not work. The SGD methods never get within the threshold, thus they go through all 5000 iterations and converge to minimal solutions which are different than the optimal solution. The sgd with the smallest learning rate never reaches a mimimal solution, it requires more iterations and takes the longest. Stochastic gradient descent with larger learning rates had higher variance, more stochastic plots. The learning rate was not large enough to leave the minimal solution which the other SGDs with smaller learning rates converge to.

Part C

SGD for the plots with constant learning rates with no solutions behave similiarly as in part b. They all approach minmial points. In this case there is not a solution. Average error with a decreasing learning rate works the best, it gets the smallest error through the iterations. It is able to break out of the non optimal mimial solutions that other SGD learning rates get stuck in.

Question 3

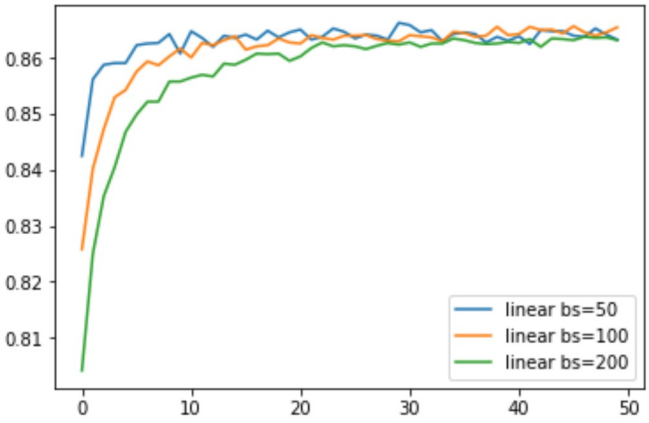
In [15]:
```
import time

import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)


def optimize(x, y, pred, loss, optimizer, training_epochs, batch_size):
    acc = []
    with tf.Session() as sess:  # start training
        sess.run(tf.global_variables_initializer())  # Run the initializer
        for epoch in range(training_epochs):  # Training cycle
            avg_loss = 0.
            total_batch = int(mnist.train.num_examples / batch_size)
            for i in range(total_batch):
                batch_xs, batch_ys = mnist.train.next_batch(batch_size)
                _, c = sess.run([optimizer, loss], feed_dict={x: batch_xs, y: batch
_ys})
                avg_loss += c / total_batch

            correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
            accuracy_ = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
            accuracy = accuracy_.eval({x: mnist.test.images, y: mnist.test.labels})
            acc.append(accuracy)
            print("Epoch:", '%04d' % (epoch + 1), "loss=", "{:.9f}".format(avg_loss
),
                  "accuracy={:.9f}".format(accuracy))
    return acc


def train_linear(learning_rate=0.01, training_epochs=50, batch_size=100):
    x = tf.placeholder(tf.float32, [None, 784])
    y = tf.placeholder(tf.float32, [None, 10])

    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))

    pred = tf.matmul(x, W) + b
    loss = tf.reduce_mean((y - pred)**2)

    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
    return optimize(x, y, pred, loss, optimizer, training_epochs, batch_size)
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

3.a

In [16]:
```
for batch_size in [50, 100, 200]:
    time_start = time.time()
    acc_linear = train_linear(batch_size=batch_size)
    print("train_linear finishes in %.3fs for batch size %s" % (time.time() - time_
start, batch_size))

    plt.plot(acc_linear, label="linear bs=%d" % batch_size)
    plt.legend()
```

```
Epoch: 0001 loss= 0.052325524 accuracy=0.842499971
Epoch: 0002 loss= 0.043814518 accuracy=0.856199980
Epoch: 0003 loss= 0.042413708 accuracy=0.858799994
Epoch: 0004 loss= 0.041666317 accuracy=0.859099984
Epoch: 0005 loss= 0.041182621 accuracy=0.859099984
Epoch: 0006 loss= 0.040828640 accuracy=0.862299979
Epoch: 0007 loss= 0.040561515 accuracy=0.862600029
Epoch: 0008 loss= 0.040346901 accuracy=0.862699986
Epoch: 0009 loss= 0.040166624 accuracy=0.864300013
Epoch: 0010 loss= 0.040022808 accuracy=0.860800028
Epoch: 0011 loss= 0.039901100 accuracy=0.864799976
Epoch: 0012 loss= 0.039787975 accuracy=0.863600016
Epoch: 0013 loss= 0.039697775 accuracy=0.861999989
Epoch: 0014 loss= 0.039616773 accuracy=0.864000022
Epoch: 0015 loss= 0.039542637 accuracy=0.863600016
Epoch: 0016 loss= 0.039470060 accuracy=0.864199996
Epoch: 0017 loss= 0.039420231 accuracy=0.863300025
Epoch: 0018 loss= 0.039365003 accuracy=0.864899993
Epoch: 0019 loss= 0.039321283 accuracy=0.863699973
Epoch: 0020 loss= 0.039277030 accuracy=0.864600003
Epoch: 0021 loss= 0.039238091 accuracy=0.865100026
Epoch: 0022 loss= 0.039198191 accuracy=0.863300025
Epoch: 0023 loss= 0.039161644 accuracy=0.863799989
Epoch: 0024 loss= 0.039133059 accuracy=0.865300000
Epoch: 0025 loss= 0.039105932 accuracy=0.864700019
Epoch: 0026 loss= 0.039074730 accuracy=0.863499999
Epoch: 0027 loss= 0.039051806 accuracy=0.864199996
Epoch: 0028 loss= 0.039025934 accuracy=0.864000022
Epoch: 0029 loss= 0.039002667 accuracy=0.863300025
Epoch: 0030 loss= 0.038985577 accuracy=0.866299987
Epoch: 0031 loss= 0.038964787 accuracy=0.865899980
Epoch: 0032 loss= 0.038938309 accuracy=0.864600003
Epoch: 0033 loss= 0.038921935 accuracy=0.865000010
Epoch: 0034 loss= 0.038909800 accuracy=0.862999976
Epoch: 0035 loss= 0.038893973 accuracy=0.864300013
Epoch: 0036 loss= 0.038879929 accuracy=0.864600003
Epoch: 0037 loss= 0.038862899 accuracy=0.864300013
Epoch: 0038 loss= 0.038851258 accuracy=0.862800002
Epoch: 0039 loss= 0.038839092 accuracy=0.863799989
Epoch: 0040 loss= 0.038827764 accuracy=0.863099992
Epoch: 0041 loss= 0.038813858 accuracy=0.863900006
Epoch: 0042 loss= 0.038802457 accuracy=0.862500012
Epoch: 0043 loss= 0.038792210 accuracy=0.865000010
Epoch: 0044 loss= 0.038779936 accuracy=0.864799976
Epoch: 0045 loss= 0.038768745 accuracy=0.864799976
Epoch: 0046 loss= 0.038756425 accuracy=0.864000022
Epoch: 0047 loss= 0.038752714 accuracy=0.863799989
Epoch: 0048 loss= 0.038742873 accuracy=0.865300000
Epoch: 0049 loss= 0.038735349 accuracy=0.864199996
Epoch: 0050 loss= 0.038725980 accuracy=0.863300025
train_linear finishes in 91.557s for batch size 50
Epoch: 0001 loss= 0.058055851 accuracy=0.825800002
Epoch: 0002 loss= 0.046460556 accuracy=0.840200007
Epoch: 0003 loss= 0.044295421 accuracy=0.847299993
Epoch: 0004 loss= 0.043251014 accuracy=0.852999985
Epoch: 0005 loss= 0.042596575 accuracy=0.854300022
Epoch: 0006 loss= 0.042127317 accuracy=0.857599974
Epoch: 0007 loss= 0.041767435 accuracy=0.859399974
Epoch: 0008 loss= 0.041477467 accuracy=0.858699977
Epoch: 0009 loss= 0.041239175 accuracy=0.860300004
Epoch: 0010 loss= 0.041035219 accuracy=0.861699998
Epoch: 0011 loss= 0.040862905 accuracy=0.860099971
Epoch: 0012 loss= 0.040707448 accuracy=0.862699986
Epoch: 0013 loss= 0.040574536 accuracy=0.862299979
```



- one epoch = one forward pass and one backward pass of all the training examples
- batch size = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of iterations = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network (https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network)

https://stats.stackexchange.com/questions/140811/how-large-should-the-batch-size-be-for-stochastic-gradient-descent (https://stats.stackexchange.com/questions/140811/how-large-should-the-batch-size-be-for-stochastic-gradient-descent)

A small batch size has more noise per step, but each training step takes less time. Smaller batch sizes results in the epoch taking longer time due to the noise.

Larger batch size has less noise per step, the gradient direction fluctuates less compared to the small batch size model. The training steps lake longer, but its more accurate. The epochs will take less time due to less noise.

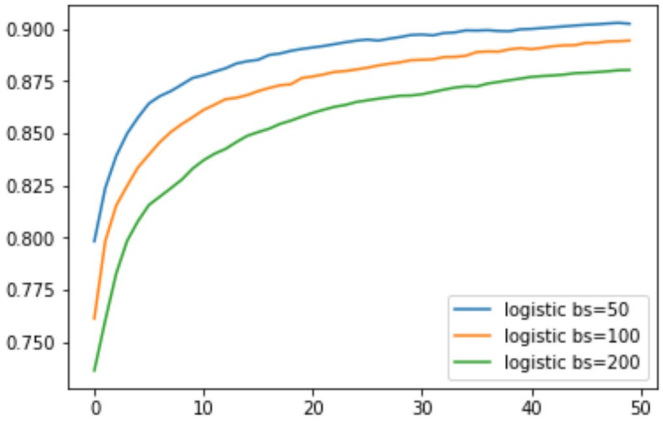
For a given accuracy/threshold smaller batch size may lead to a shorter total training time.

3.b

In [17]:
```
def softmax(z):
    z_ = tf.reduce_max(z)
    return tf.exp(z - z_) / tf.reduce_sum(tf.exp(z - z_))

def train_logistic(learning_rate=0.01, training_epochs=50, batch_size=100):
    x = tf.placeholder(tf.float32, [None, 784])
    y = tf.placeholder(tf.float32, [None, 10])

    W = tf.Variable(tf.zeros([784, 10]))
    b = tf.Variable(tf.zeros([10]))

    # YOUR CODE HERE
    pred = softmax(tf.matmul(x, W) + b)
    loss = -tf.reduce_mean(y*tf.log(pred))
    ###############

    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
    return optimize(x, y, pred, loss, optimizer, training_epochs, batch_size)
```

In [18]:
```
for batch_size in [50, 100, 200]:
    time_start = time.time()
    acc_logistic = train_logistic(batch_size=batch_size)
    print("train_logistic finishes in %.3fs for batch size %s" % (time.time() - tim
e_start, batch_size))

    plt.plot(acc_logistic, label="logistic bs=%d" % batch_size)
    plt.legend()
```

```
Epoch: 0001 loss= 0.576622326 accuracy=0.798200011
Epoch: 0002 loss= 0.524414344 accuracy=0.823700011
Epoch: 0003 loss= 0.502109293 accuracy=0.838999987
Epoch: 0004 loss= 0.491084855 accuracy=0.849799991
Epoch: 0005 loss= 0.484569670 accuracy=0.857500017
Epoch: 0006 loss= 0.480181149 accuracy=0.864199996
Epoch: 0007 loss= 0.476930795 accuracy=0.867699981
Epoch: 0008 loss= 0.474394500 accuracy=0.870100021
Epoch: 0009 loss= 0.472353866 accuracy=0.873199999
Epoch: 0010 loss= 0.470655275 accuracy=0.876399994
Epoch: 0011 loss= 0.469196189 accuracy=0.877699971
Epoch: 0012 loss= 0.467964755 accuracy=0.879400015
Epoch: 0013 loss= 0.466907430 accuracy=0.880999982
Epoch: 0014 loss= 0.465927209 accuracy=0.883300006
Epoch: 0015 loss= 0.465104084 accuracy=0.884400010
Epoch: 0016 loss= 0.464342323 accuracy=0.885100007
Epoch: 0017 loss= 0.463675702 accuracy=0.887399971
Epoch: 0018 loss= 0.463040211 accuracy=0.888100028
Epoch: 0019 loss= 0.462474291 accuracy=0.889400005
Epoch: 0020 loss= 0.461955960 accuracy=0.890299976
Epoch: 0021 loss= 0.461463005 accuracy=0.890999973
Epoch: 0022 loss= 0.461008052 accuracy=0.891700029
Epoch: 0023 loss= 0.460600740 accuracy=0.892600000
Epoch: 0024 loss= 0.460215388 accuracy=0.893499970
Epoch: 0025 loss= 0.459841508 accuracy=0.894299984
Epoch: 0026 loss= 0.459510202 accuracy=0.894800007
Epoch: 0027 loss= 0.459189591 accuracy=0.894299984
Epoch: 0028 loss= 0.458891316 accuracy=0.895200014
Epoch: 0029 loss= 0.458588090 accuracy=0.896000028
Epoch: 0030 loss= 0.458324805 accuracy=0.897000015
Epoch: 0031 loss= 0.458071958 accuracy=0.897199988
Epoch: 0032 loss= 0.457844734 accuracy=0.896799982
Epoch: 0033 loss= 0.457603747 accuracy=0.897899985
Epoch: 0034 loss= 0.457365061 accuracy=0.898199975
Epoch: 0035 loss= 0.457159536 accuracy=0.899200022
Epoch: 0036 loss= 0.456947370 accuracy=0.899100006
Epoch: 0037 loss= 0.456760338 accuracy=0.899299979
Epoch: 0038 loss= 0.456583263 accuracy=0.898899972
Epoch: 0039 loss= 0.456383456 accuracy=0.898800015
Epoch: 0040 loss= 0.456202655 accuracy=0.899699986
Epoch: 0041 loss= 0.456069838 accuracy=0.899900019
Epoch: 0042 loss= 0.455899679 accuracy=0.900300026
Epoch: 0043 loss= 0.455745235 accuracy=0.900699973
Epoch: 0044 loss= 0.455596625 accuracy=0.901099980
Epoch: 0045 loss= 0.455430753 accuracy=0.901499987
Epoch: 0046 loss= 0.455318729 accuracy=0.901899993
Epoch: 0047 loss= 0.455187068 accuracy=0.902100027
Epoch: 0048 loss= 0.455060951 accuracy=0.902499974
Epoch: 0049 loss= 0.454895884 accuracy=0.902800024
Epoch: 0050 loss= 0.454814552 accuracy=0.902400017
train_logistic finishes in 90.297s for batch size 50
Epoch: 0001 loss= 0.665247361 accuracy=0.761200011
Epoch: 0002 loss= 0.626570524 accuracy=0.798200011
Epoch: 0003 loss= 0.601731353 accuracy=0.815299988
Epoch: 0004 loss= 0.585722472 accuracy=0.824699998
Epoch: 0005 loss= 0.575122531 accuracy=0.833599985
Epoch: 0006 loss= 0.567796781 accuracy=0.839699984
Epoch: 0007 loss= 0.562471698 accuracy=0.845700026
Epoch: 0008 loss= 0.558443743 accuracy=0.850499988
Epoch: 0009 loss= 0.555264835 accuracy=0.854300022
Epoch: 0010 loss= 0.552675713 accuracy=0.857599974
Epoch: 0011 loss= 0.550492055 accuracy=0.861100018
Epoch: 0012 loss= 0.548637444 accuracy=0.863600016
Epoch: 0013 loss= 0.547040470 accuracy=0.866199970
```

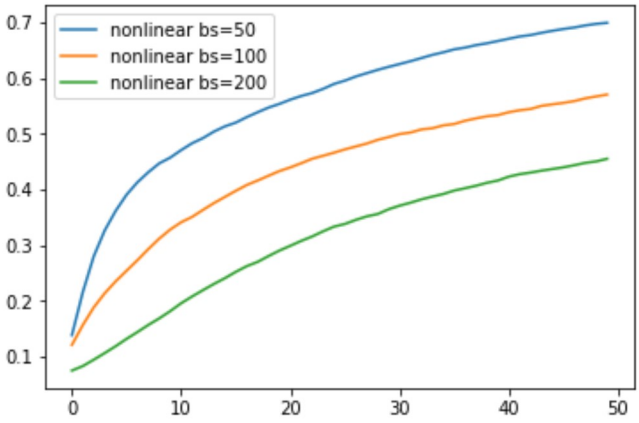


3.c

In [19]:
```
def train_nn(learning_rate=0.01, training_epochs=50, batch_size=50, n_hidden=64):
    x = tf.placeholder(tf.float32, [None, 784])
    y = tf.placeholder(tf.float32, [None, 10])

    W1 = tf.Variable(tf.random_normal([784, n_hidden]))
    W2 = tf.Variable(tf.random_normal([n_hidden, 10]))
    b1 = tf.Variable(tf.random_normal([n_hidden]))
    b2 = tf.Variable(tf.random_normal([10]))

    # YOUR CODE HERE
    pred = softmax(tf.matmul(tf.tanh(tf.matmul(x, W1) + b1), W2) + b2)
    loss = -tf.reduce_mean(y*tf.log(pred))
    ################

    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
    return optimize(x, y, pred, loss, optimizer, training_epochs, batch_size)
```

In [20]:
```
for batch_size in [50, 100, 200]:
    time_start = time.time()
    acc_nn = train_nn(batch_size=batch_size)
    print("train_nn finishes in %.3fs for batch size %s" % (time.time() - time_star
t, batch_size))

    plt.plot(acc_nn, label="nonlinear bs=%d" % batch_size)
    plt.legend()
```

```
Epoch: 0001 loss= 1.826924652 accuracy=0.139100000
Epoch: 0002 loss= 1.475448279 accuracy=0.216399997
Epoch: 0003 loss= 1.290024038 accuracy=0.280600011
Epoch: 0004 loss= 1.165667411 accuracy=0.326900005
Epoch: 0005 loss= 1.072994833 accuracy=0.361999989
Epoch: 0006 loss= 1.004285814 accuracy=0.391099989
Epoch: 0007 loss= 0.950219669 accuracy=0.413500011
Epoch: 0008 loss= 0.904089302 accuracy=0.431600004
Epoch: 0009 loss= 0.867574533 accuracy=0.447499990
Epoch: 0010 loss= 0.834793732 accuracy=0.457700014
Epoch: 0011 loss= 0.805960297 accuracy=0.471199989
Epoch: 0012 loss= 0.781864277 accuracy=0.483500004
Epoch: 0013 loss= 0.759655047 accuracy=0.492799997
Epoch: 0014 loss= 0.738980524 accuracy=0.504499972
Epoch: 0015 loss= 0.720648123 accuracy=0.513800025
Epoch: 0016 loss= 0.704221573 accuracy=0.520600021
Epoch: 0017 loss= 0.689879290 accuracy=0.530399978
Epoch: 0018 loss= 0.675091906 accuracy=0.539200008
Epoch: 0019 loss= 0.663065379 accuracy=0.547599971
Epoch: 0020 loss= 0.651334806 accuracy=0.554199994
Epoch: 0021 loss= 0.641135833 accuracy=0.561900020
Epoch: 0022 loss= 0.631374717 accuracy=0.568400025
Epoch: 0023 loss= 0.622168692 accuracy=0.573700011
Epoch: 0024 loss= 0.613869152 accuracy=0.581200004
Epoch: 0025 loss= 0.606178266 accuracy=0.589999974
Epoch: 0026 loss= 0.599183296 accuracy=0.596000016
Epoch: 0027 loss= 0.592465262 accuracy=0.603399992
Epoch: 0028 loss= 0.586603195 accuracy=0.609399974
Epoch: 0029 loss= 0.580731180 accuracy=0.615300000
Epoch: 0030 loss= 0.575674135 accuracy=0.620700002
Epoch: 0031 loss= 0.570846911 accuracy=0.625699997
Epoch: 0032 loss= 0.566258838 accuracy=0.630900025
Epoch: 0033 loss= 0.562061518 accuracy=0.636500001
Epoch: 0034 loss= 0.558200832 accuracy=0.642199993
Epoch: 0035 loss= 0.554560828 accuracy=0.646799982
Epoch: 0036 loss= 0.551112570 accuracy=0.652100027
Epoch: 0037 loss= 0.547903235 accuracy=0.655300021
Epoch: 0038 loss= 0.544948475 accuracy=0.659699976
Epoch: 0039 loss= 0.541963240 accuracy=0.662999988
Epoch: 0040 loss= 0.539366073 accuracy=0.667100012
Epoch: 0041 loss= 0.536840622 accuracy=0.671199977
Epoch: 0042 loss= 0.534490673 accuracy=0.675300002
Epoch: 0043 loss= 0.532196962 accuracy=0.677800000
Epoch: 0044 loss= 0.530076861 accuracy=0.681900024
Epoch: 0045 loss= 0.528099804 accuracy=0.685500026
Epoch: 0046 loss= 0.526145202 accuracy=0.688700020
Epoch: 0047 loss= 0.524464198 accuracy=0.691399992
Epoch: 0048 loss= 0.522706910 accuracy=0.695100009
Epoch: 0049 loss= 0.521084750 accuracy=0.697899997
Epoch: 0050 loss= 0.519572063 accuracy=0.699699998
train_nn finishes in 55.898s for batch size 50
Epoch: 0001 loss= 2.094550627 accuracy=0.121500000
Epoch: 0002 loss= 1.837426821 accuracy=0.157199994
Epoch: 0003 loss= 1.671983317 accuracy=0.188899994
Epoch: 0004 loss= 1.558028332 accuracy=0.214100003
Epoch: 0005 loss= 1.465197563 accuracy=0.235499993
Epoch: 0006 loss= 1.389495982 accuracy=0.254599988
Epoch: 0007 loss= 1.328111940 accuracy=0.273600012
Epoch: 0008 loss= 1.274340292 accuracy=0.293599993
Epoch: 0009 loss= 1.224440856 accuracy=0.312400013
Epoch: 0010 loss= 1.181753323 accuracy=0.328599989
Epoch: 0011 loss= 1.145634513 accuracy=0.341699988
Epoch: 0012 loss= 1.111235581 accuracy=0.351300001
Epoch: 0013 loss= 1.080213286 accuracy=0.363900006
```


nonlinear bs=50
nonlinear bs=100
nonlinear bs=200
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50

3.e

In [21]:
```
import numpy as np

n_data = 6000
n_dim = 50

w_true = np.random.uniform(low=-2.0, high=2.0, size=[n_dim])

x_true = np.random.uniform(low=-10.0, high=10.0, size=[n_data, n_dim])
x_ob = x_true + np.random.randn(n_data, n_dim)
y_ob = x_true @ w_true + np.random.randn(n_data)

learning_rate = 0.0001
training_epochs = 100
batch_size = 6000


def main():
    x = tf.placeholder(tf.float32, [None, n_dim])
    y = tf.placeholder(tf.float32, [None, 1])

    w = tf.Variable(tf.random_normal([n_dim, 1]))

    # YOUR CODE HERE
    likelihood = -(n_data/2)*tf.log(tf.square(tf.norm(w)) + 1) - tf.square(tf.norm(
y-tf.matmul(x, w))) / (2*(tf.square(tf.norm(w)) + 1))
    cost = -1.0 * likelihood
    ################

    # Adam is a fancier version of SGD, which is insensitive to the learning
    # rate.  Try replace this with GradientDescentOptimizer and tune the
    # parameters!
    optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        w_sgd = sess.run(w).flatten()

        for epoch in range(training_epochs):
            avg_cost = 0.
            total_batch = int(n_data / batch_size)
            for i in range(total_batch):
                start, end = i * batch_size, (i + 1) * batch_size
                _, c = sess.run(
                    [optimizer, cost],
                    feed_dict={
                        x: x_ob[start:end, :],
                        y: y_ob[start:end, np.newaxis]
                    })
                avg_cost += c / total_batch
            w_sgd = sess.run(w).flatten()
            print("Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(avg_cost
),
                  "|w-w_true|^2 = {:.9f}".format(np.sum((w_sgd - w_true)**2)))

    # Total least squares: SVD
    X = x_true
    y = y_ob
    stacked_mat = np.hstack((X, y[:, np.newaxis])).astype(np.float32)
    u, s, vh = np.linalg.svd(stacked_mat)
    w_tls = -vh[-1, :-1] / vh[-1, -1]

    error = np.sum(np.square(w_tls - w_true))
    print("TLS through SVD error: |w-w_true|^2 = {}".format(error))
```

```
Epoch: 0001 cost= 253665.390625000 |w-w_true|^2 = 125.327405754
Epoch: 0002 cost= 117488.406250000 |w-w_true|^2 = 116.827041492
Epoch: 0003 cost= 104362.882812500 |w-w_true|^2 = 107.489208144
Epoch: 0004 cost= 94190.070312500 |w-w_true|^2 = 97.838765985
Epoch: 0005 cost= 85752.265625000 |w-w_true|^2 = 88.159607038
Epoch: 0006 cost= 78450.000000000 |w-w_true|^2 = 78.630296506
Epoch: 0007 cost= 71944.078125000 |w-w_true|^2 = 69.375924598
Epoch: 0008 cost= 66025.187500000 |w-w_true|^2 = 60.491753193
Epoch: 0009 cost= 60556.691406250 |w-w_true|^2 = 52.055214104
Epoch: 0010 cost= 55446.718750000 |w-w_true|^2 = 44.132340809
Epoch: 0011 cost= 50633.492187500 |w-w_true|^2 = 36.781214116
Epoch: 0012 cost= 46077.359375000 |w-w_true|^2 = 30.053478457
Epoch: 0013 cost= 41756.554687500 |w-w_true|^2 = 23.994389795
Epoch: 0014 cost= 37665.257812500 |w-w_true|^2 = 18.641536914
Epoch: 0015 cost= 33812.867187500 |w-w_true|^2 = 14.022247105
Epoch: 0016 cost= 30223.738281250 |w-w_true|^2 = 10.149629984
Epoch: 0017 cost= 26936.316406250 |w-w_true|^2 = 7.017514727
Epoch: 0018 cost= 24000.195312500 |w-w_true|^2 = 4.595000541
Epoch: 0019 cost= 21469.382812500 |w-w_true|^2 = 2.822270060
Epoch: 0020 cost= 19390.281250000 |w-w_true|^2 = 1.610226590
Epoch: 0021 cost= 17785.412109375 |w-w_true|^2 = 0.846705138
Epoch: 0022 cost= 16638.136718750 |w-w_true|^2 = 0.410118038
Epoch: 0023 cost= 15887.690429688 |w-w_true|^2 = 0.187229185
Epoch: 0024 cost= 15441.373046875 |w-w_true|^2 = 0.088031973
Epoch: 0025 cost= 15199.513671875 |w-w_true|^2 = 0.051643969
Epoch: 0026 cost= 15078.782226562 |w-w_true|^2 = 0.042881991
Epoch: 0027 cost= 15022.316406250 |w-w_true|^2 = 0.044231517
Epoch: 0028 cost= 14997.099609375 |w-w_true|^2 = 0.048159992
Epoch: 0029 cost= 14986.165039062 |w-w_true|^2 = 0.051941059
Epoch: 0030 cost= 14981.500976562 |w-w_true|^2 = 0.054857124
Epoch: 0031 cost= 14979.527343750 |w-w_true|^2 = 0.056907081
Epoch: 0032 cost= 14978.695312500 |w-w_true|^2 = 0.058282071
Epoch: 0033 cost= 14978.342773438 |w-w_true|^2 = 0.059180917
Epoch: 0034 cost= 14978.193359375 |w-w_true|^2 = 0.059760118
Epoch: 0035 cost= 14978.130859375 |w-w_true|^2 = 0.060130438
Epoch: 0036 cost= 14978.102539062 |w-w_true|^2 = 0.060366313
Epoch: 0037 cost= 14978.091796875 |w-w_true|^2 = 0.060516240
Epoch: 0038 cost= 14978.085937500 |w-w_true|^2 = 0.060611581
Epoch: 0039 cost= 14978.084960938 |w-w_true|^2 = 0.060672275
Epoch: 0040 cost= 14978.083984375 |w-w_true|^2 = 0.060710912
Epoch: 0041 cost= 14978.083007812 |w-w_true|^2 = 0.060735594
Epoch: 0042 cost= 14978.083007812 |w-w_true|^2 = 0.060751410
Epoch: 0043 cost= 14978.082031250 |w-w_true|^2 = 0.060761500
Epoch: 0044 cost= 14978.083007812 |w-w_true|^2 = 0.060767990
Epoch: 0045 cost= 14978.083007812 |w-w_true|^2 = 0.060772163
Epoch: 0046 cost= 14978.082031250 |w-w_true|^2 = 0.060774884
Epoch: 0047 cost= 14978.083007812 |w-w_true|^2 = 0.060776607
Epoch: 0048 cost= 14978.083984375 |w-w_true|^2 = 0.060777682
Epoch: 0049 cost= 14978.083984375 |w-w_true|^2 = 0.060778405
Epoch: 0050 cost= 14978.083984375 |w-w_true|^2 = 0.060778868
Epoch: 0051 cost= 14978.083007812 |w-w_true|^2 = 0.060779161
Epoch: 0052 cost= 14978.083984375 |w-w_true|^2 = 0.060779372
Epoch: 0053 cost= 14978.083984375 |w-w_true|^2 = 0.060779511
Epoch: 0054 cost= 14978.083984375 |w-w_true|^2 = 0.060779578
Epoch: 0055 cost= 14978.083007812 |w-w_true|^2 = 0.060779660
Epoch: 0056 cost= 14978.082031250 |w-w_true|^2 = 0.060779695
Epoch: 0057 cost= 14978.082031250 |w-w_true|^2 = 0.060779701
Epoch: 0058 cost= 14978.082031250 |w-w_true|^2 = 0.060779716
Epoch: 0059 cost= 14978.082031250 |w-w_true|^2 = 0.060779724
Epoch: 0060 cost= 14978.082031250 |w-w_true|^2 = 0.060779731
Epoch: 0061 cost= 14978.082031250 |w-w_true|^2 = 0.060779732
Epoch: 0062 cost= 14978.082031250 |w-w_true|^2 = 0.060779746
Epoch: 0063 cost= 14978.082031250 |w-w_true|^2 = 0.060779746
Epoch: 0064 cost= 14978.082031250 |w-w_true|^2 = 0.060779746
```

The solutions are sensitive to the hyperparamaters. Regular gradient descent is the most costly but achieves the best results. Batch size and learning rate have the largest role.

Question 4

PDF MERGED BELOW:

In [4]:
```
import numpy as np
import scipy as sp
import scipy.stats as st
import scipy.interpolate
import scipy.linalg as la
import pylab as pl
import math
from scipy import stats
from scipy.stats import multivariate_normal, probplot
from sklearn import preprocessing
import statsmodels.api as sm
import pickle

import scipy as SP
import scipy.optimize as opt
import pylab
import matplotlib.pyplot as plt
plt.style.use("fivethirtyeight")

# import helper functions from starter code
from gwas import *

%matplotlib inline
```

```
/usr/local/lib/python3.5/dist-packages/statsmodels/compat/pandas.py:56: FutureWa
rning: The pandas.core.datetools module is deprecated and will be removed in a f
uture version. Please use the pandas.tseries module instead.
  from pandas.core import datetools
```

GWAS problem - Introduction

Overall goal: This real world problem is one in computational biology which uses many of the techniques and concepts you have been introduced to, all together, in particular, linear regression, PCA, non-iid noise, diagonalizing multivariate Gaussian covariance matrices, and bias-variance trade-off. We will also tangentially introduce you to concepts of statistical testing. \textbf{This homework problem is effectively a demo in that we will ask you to execute code and answer questions about what you observe. You are not required to code anything at all.}

Setup and problem statement: Given a set of people for whom genetics (DNA) has been measured, and also a corresponding trait for each person, such as blood pressure, or "is-a-smoker", one can use data-driven methods to deduce which genetic effects are likely responsible for the trait. We have collected blood from n individuals who either smoke ($y_i = 1$) or do not smoke ($y_i = 0$). Their blood samples have been sequenced at m positions along the genome, yielding the feature matrix $X \in \mathbb{R}^{n \times m}$, composed of the genetic variants (which take on values 0, 1, or 2). Specifically, $X_{i,j}$ is a numeric encoding of the DNA for the ith person at genetic feature j. We want to deduce which of the m genetic features are associated with smoking, by considering one at a time.

In the data we give you, it will turn out that there is no true signal in the data; however, we will see that without careful modeling, we would erroneously conclude that the data was full of signal. Left unmodeled, such structure creates misleading results, yielding signal where none exists.

Overall modelling approach: The basic idea will be to "test" one genetic feature at a time and assign a score (a p-value) indicating our level of belief that it is associated with the trait. We will start with a simple linear regression model, and then build increasingly more correct linear predictive models. The three models we will use are (1) linear regression, (2) linear regression with PCA features, (3) linear regression with all genetic variants as features. The first model is naive, because it assumes the individuals are iid. The fundamental modelling challenges with these kinds of analyses is that the individuals in the study are not typically not iid, owing to differences in race and family-relatedness (e.g., sisters, brothers, patients, grandparents in the data set), which violate the iid assumption. Left unmodelled, such structure creates misleading results, yielding signal where none exists. Luckily, as we shall see, one can visualize these modelling issues via quantile-quantile plots, which will soon be briefly introduced.

Quick introduction to Hypothesis Testing

To understand whether genetic marker j is informative in predicting the the target, we will set up a hypothesis test. A Hypothesis Test allows us to examine two different "views" of the world, the Null and the Alternative, and conclude which one is more likely based on the data we observe. A Hypothesis Test has the following main components:

- **Null Hypothesis:** The genetic marker is not significant in predicting the target. Any relation you observe between the genetic marker and the target is merely due to chance.
- **Alternative Hypothesis:** The genetic marker contains important information about the target.
- **p-value:** The final result of a hypothesis test is a p-value. A p-value is a number between $[0, 1]$ which you should interpret as follows: If the p-value is really small, that is evidence against the Null and in favor of the Alternative hypothesis. If the p-value is large then this is evidence towards the Null.

We will perform a single Hypothesis Test for each of the genetic markers that we have in our dataset. This process, will give us back m p-values. These m values will make up the empirical distribution of our p-values.

How to test each variant Herein we provide a minimal exposition to allow you to do the homework. To estimate how implicated each genetic feature is we will use a score, in the form of a p-value. One can think of the p-value as a proxy for how informative the genetic feature is for prediction of the trait (e.g. "is smoker"). More precisely, to get the p-value for the j^{th} genetic feature, we first include it in the model (for a given model class, such as linear regression) and compute the maximum likelihood, LL_j (this is our alternative hypothesis). Then we repeat this process, but having removed the genetic feature of interest from the model, yielding LL_{-j} (this is our null hypothesis). To be clear, the null hypothesis will have none of the m genetic variants that are being tested. You can refer to the jupyter notebook for a brief explanation of hypothesis testing. The p-value is then a simple monotonic decreasing function of the difference in these two likelihoods, `diff_ll` $= LL_j - LL_{-j}$--one that we will give you. P-values lie in $[0, 1]$ and the smaller the p-value, the larger the difference in the likelihoods, and the more evidence that the genetic marker is associated with the trait (assuming the model is correct).

Part a

To diagnose if something is amiss in our genetic analyses, we will make use of the following: (1) we assume that if any genetic signal is present, it is restricted to a small fraction of the genetic features, (2) p-values corresponding to no signal in the data are distributed as $p \sim \text{Unif}[0, 1]$. Combining these two assumptions, we will assume that p-values arising from a valid model should largely be drawn from $\text{Unif}[0, 1]$, and also that large deviations suggest that our model is incorrect. Quantile-quantile plots let us visualize if we have deviations or not. Quantile-quantile plots are a way to compare two probability distributions by comparing their quantile values (e.g. how does the smallest p-value in each distribution compare, and then the second smallest, etc.). In the quantile-quantile plot, you will see m points, one for each genetic marker. The x-coordinate of each point corresponds to the theoretical quantile we would expect to see if the distribution was in fact a $Unif[0, 1]$ and the y-coordinate corresponds to the observed value of the quantile. An example is shown in Figure 1, where the line on the diagonal results from an analysis where the model is correct, and hence the theoretical and empirical p-value quantiles match, while the other line, which deviates from the diagonal, indicates that we have likely made a modelling error. If there are genetic signals in the data, these would simply emerge as a handful of outlier points from the diagonal (not shown).

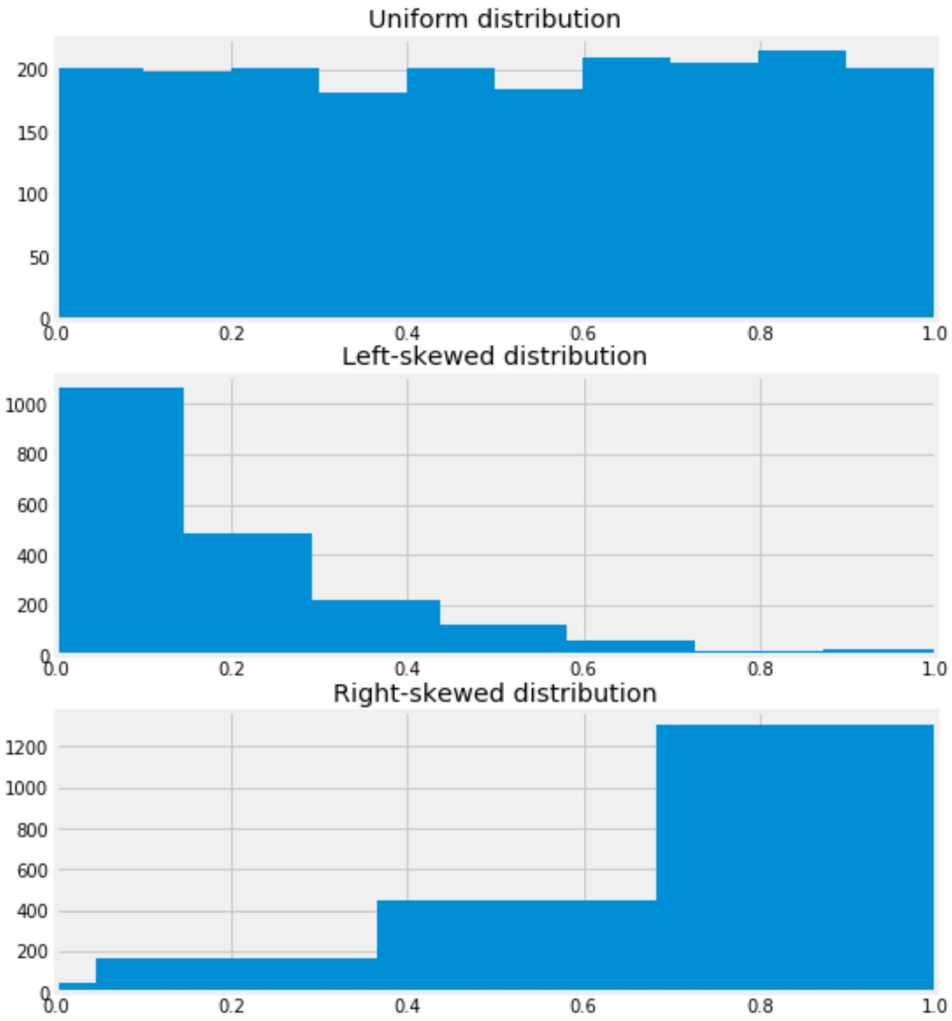
Before we dive into developing our models, we need to be able to understand whether the p-values we get back from our hypothesis tests look like m random draws from a $Unif[0, 1]$.

Use the `qqplot` function to make a qq-plot for each of the 3 distributions provided below and explain your findings. What should we observe in our qq-plot if our empirical distribution looks more and more similar to a Unif[0, 1]? Note that we use two kinds of qq-plots: one in p-value space and one negative log p-value space. The former is for intuition, while the latter is for higher resolution. The green lines in the negative log p-value qq-plots indicate the error bars.

In [2]:
```
# generate 3 distributions
n_points =  2000
unif = np.random.uniform(0, 1, n_points)
skewed_left = np.random.exponential(scale=0.2, size=n_points)
skewed_right = 1-np.random.exponential(scale=0.3, size=n_points)
```

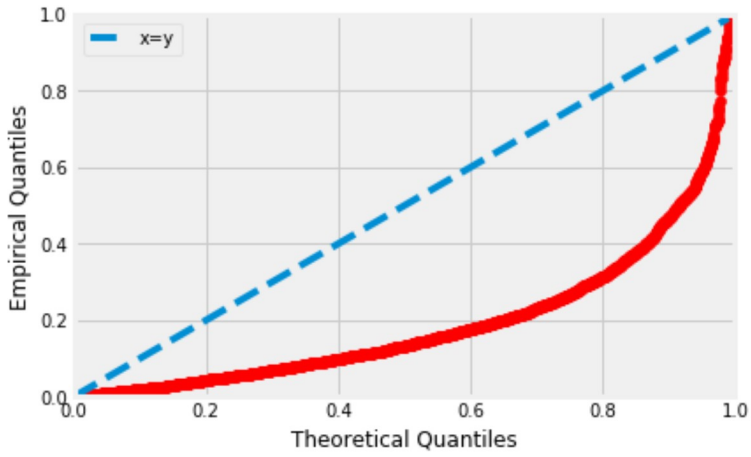
In [3]:
```
fig, axes = plt.subplots(nrows=3, ncols=1)
fig.set_figheight(10)
fig.set_figwidth(8)

# first subplot
axes[0].hist(unif)
axes[0].set_title('Uniform distribution')
axes[0].set_xlim(0, 1)

# second subplot
axes[1].hist(skewed_left)
axes[1].set_title('Left-skewed distribution')
axes[1].set_xlim(0, 1)

# third subplot
axes[2].hist(skewed_right)
axes[2].set_title('Right-skewed distribution')
axes[2].set_xlim(0, 1)
```

Out[3]: (0, 1)

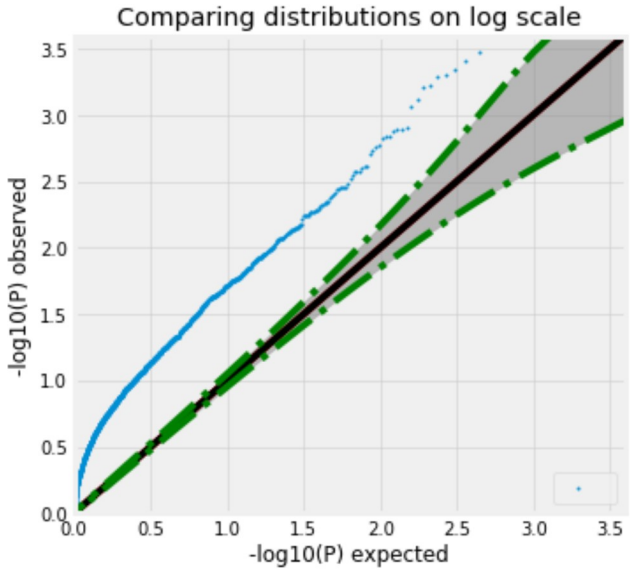


In [51]:
```
def qqplot(empirical_dist, legend = ""):
    """
    Generates two qq-plots one in the original scale of the data and
    one using a negative log transformation to make the difference
    between the expected and actual behavior more visible.

    If the observed line is outside the grey region denoted by the error bars,
    then it is quite unlikely that our data came from a Unif[0, 1] distribution.
    """
    x, y = probplot(empirical_dist, dist=stats.distributions.uniform(), fit=False)
    plt.scatter(x, y, c='r', alpha=1, )
    plt.xlabel("")
    plt.ylabel("")
    plt.xlim(0, 1)
    plt.ylim(0, 1)
    plt.xlabel("Theoretical Quantiles")
    plt.ylabel("Empirical Quantiles")
    plt.plot(np.arange(0, 1, 0.01), np.arange(0, 1, 0.01), "--", label='x=y')
    plt.legend()
    plt.show()
    fastlmm_qqplot(empirical_dist, legend=legend, xlim=(0,3.6), ylim=(0,3.6), fixax
es=False)
```

In [52]:
```
# SOLUTION CELL
qqplot(skewed_left)
```




```
/Users/janeyu/anaconda/lib/python3.6/site-packages/numpy/lib/scimath.py:120: RuntimeWarning: invalid value encountered in less
  if any(isreal(x) & (x < 0)):
```



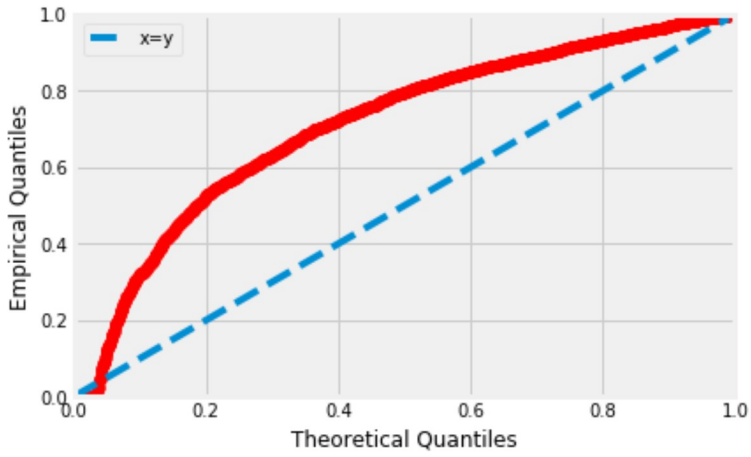
Blue line - observed line. Outside of the grey region denoted by the error bars, thus it is unlikely that our data came from a uniform distribution.

This is observed in the plot above, therefore this left_skewed data set represents a false null hypothesis. The distribution of the p value is more weighted towards higher p values.

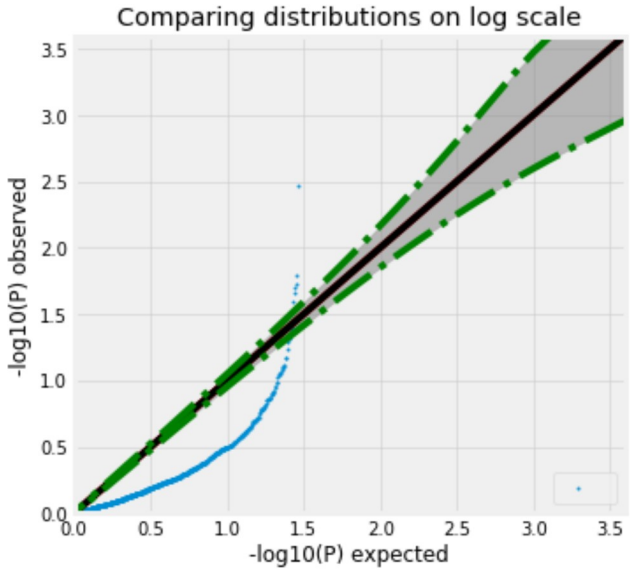
The t paramter, from our empirical distribution, is being drawn from a skewed left distribution therefore certain values are less likeley.

We should observe data that fits the uniform[0,1] distribution if our null hypothesis is true and all other assumptions are met.

In [53]:
```
# SOLUTION CELL
qqplot(skewed_right)
```




```
/Users/janeyu/anaconda/lib/python3.6/site-packages/numpy/lib/scimath.py:120: RuntimeWarning: invalid value encountered in less
  if any(isreal(x) & (x < 0)):
```



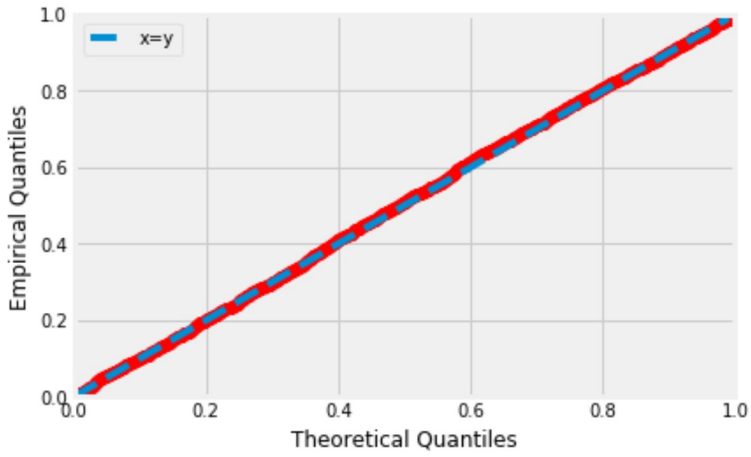
Blue line - observed line. Outside of the grey region denoted by the error bars, thus it is unlikely that our data came from a uniform distribution.

This is observed in the plot above, therefore this right_skewed data set represents a false null hypothesis. The distribution of the p value is more weighted towards zero.

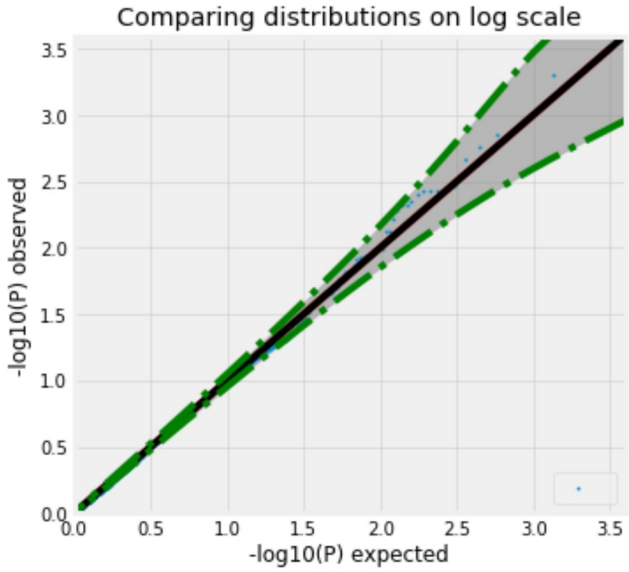
The t paramter, from our empirical distribution, is being drawn from a skewed right distribution therefore certain values are less likeley.

We should observe data that fits the uniform[0,1] distribution if our null hypothesis is true and all other assumptions are met.

In [54]:
```
# SOLUTION CELL
qqplot(unif)
```




```
/Users/janeyu/anaconda/lib/python3.6/site-packages/numpy/lib/scimath.py:120: RuntimeWarning: invalid value encountered in less
  if any(isreal(x) & (x < 0)):
```



Blue line - observed line. Observed line is within the grey region denoted by the error bars, thus it is likely that our data came from a uniform distribution. The value is uniformly distributed when the null hypothesis is true and all other assumptions are met. By definition, we want the probabilty of rejecting the true null hypothesis to be α, the null hypothesis is rejected when the observed p value is less than α

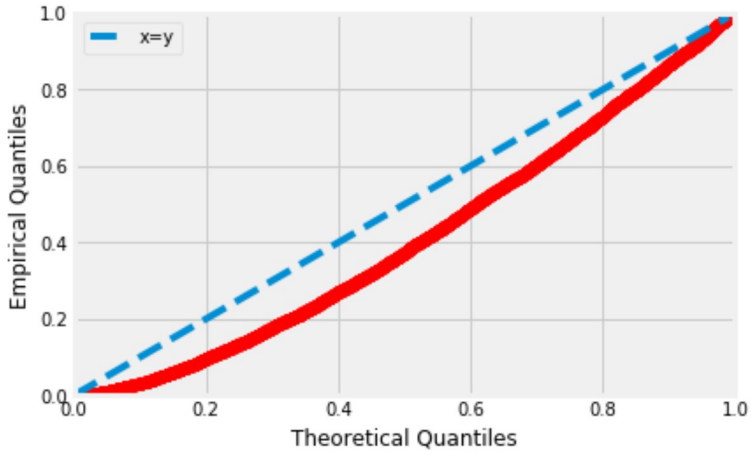
The t paramter, from our empirical distribution, is being drawn from a uniform distribution therefore all the values are equally as likely.

Part b

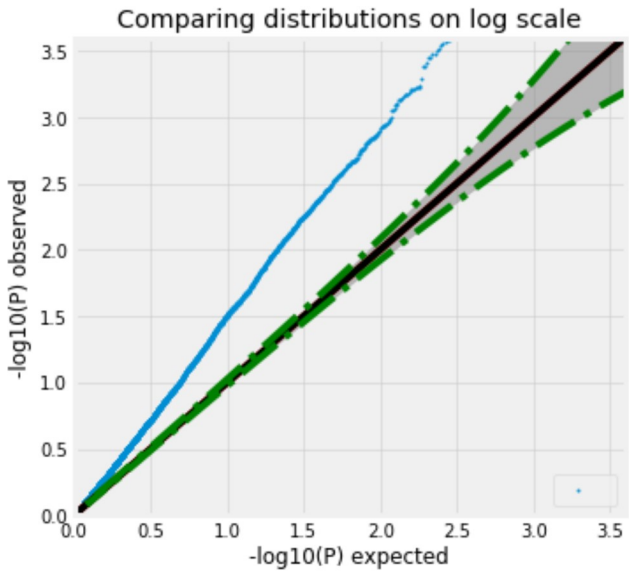
We will use linear models in the genetic marker we are testing. In particular, when testing the j^{th} genetic feature, we have that the trait, $\vec{y}$, is a linear function of the genetic variant, i.e. $\vec{y} = \overrightarrow{x_j} w_1 + \overrightarrow{w_0} + \vec{\epsilon}$ where each entry ϵ_i of ϵ is random noise distributed as $N(0, \sigma^2)$, $w_1 \in \mathbb{R}^1$, $\overrightarrow{w_0} \in \mathbb{R}^{n \times 1}$ is a constant vector, and $\overrightarrow{x_j}$ is the jth column of X, representing data for the jth genetic variant. To simplify matters, we will add a column of ones to the right end of $\overrightarrow{x_j}$ and rewrite the regression as $\vec{y} = [\overrightarrow{x_j}, \vec{1}]\vec{w} + \vec{\epsilon}$ where $[\overrightarrow{x_j}, \vec{1}]$ is the jth column of X with a vector of ones appended to the end and $\vec{w} \in \mathbb{R}^{2 \times 1}$. The model without any genetic information, $\vec{y} = \vec{1}w + \vec{\epsilon}$ is referred to as the **null model** in the parlance of statistical testing. The **alternative model**, which includes the information we are testing (one genetic marker) is $\vec{y} = [\overrightarrow{x_j}, \vec{1}]\vec{w} + \vec{\epsilon}$ where $\overrightarrow{x_j}$ is the jth column of X, i.e. the data using only the jth genetic variant as a feature. **Plot the quantile-quantile plot of p-values using linear regression as just described, a so-called naive approach, by running the function $naive\_model$. From the plot, what do you conclude about the suitability of linear regression for this problem?**

In [8]:
```
# import all the data
X = load_X()
y=load_y()

#Normalize the columns
X = preprocessing.normalize(X, norm='l2', axis = 0)
y = y.reshape(-1, 1)
```

In [55]:
```
# SOLUTION CELL
# Note that the function naive_model returns the empirical distribution of p-values
computed by the naive model
# Green lines indicate error bars.
naive_pvals = naive_model(X,y)
plt.show()
qqplot(naive_pvals)
```



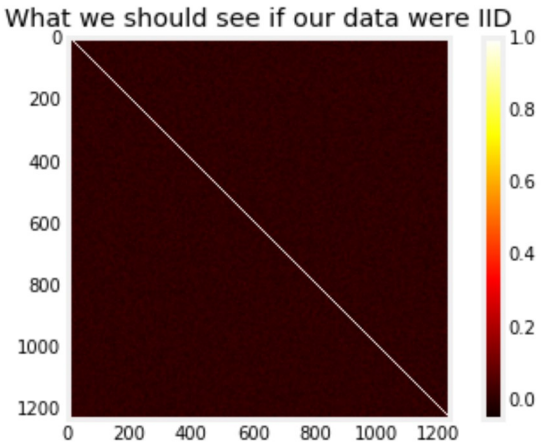

```
/Users/janeyu/anaconda/lib/python3.6/site-packages/numpy/lib/scimath.py:120: Run
timeWarning: invalid value encountered in less
  if any(isreal(x) & (x < 0)):
```



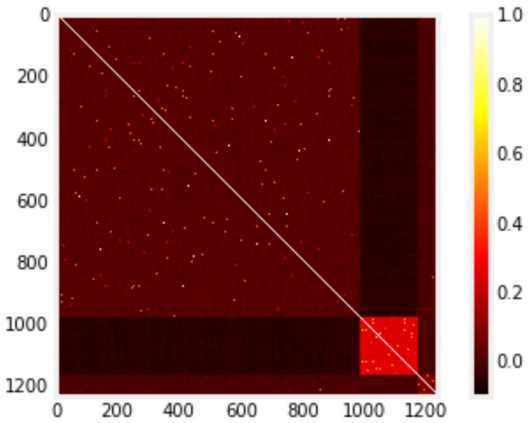
Based on the previous plots, we see that the model tells us that there are more genetic markers that are informative in predicting trait y than we would expect. Let's examine to what extent this is merely an artifact of the noise in our dataset not being iid using a correlation matrix. A correlation matrix visualizes the correlations between all datapoints in our dataset. If we can observe large correlations in the off-diagonal entries, then this presents strong evidence that our noise might not be iid.

In [10]:
```
# If our data were iid we would expect to see a correlation matrix to look somethin
g like this
X_r = np.random.randn(X.shape[0], X.shape[1])

corr_matrix = np.corrcoef(X_r)
plt.grid(False)
plt.imshow(corr_matrix, "hot")
plt.colorbar()
plt.title("What we should see if our data were IID")
```

Out[10]:
```
Text(0.5,1,'What we should see if our data were IID')
```



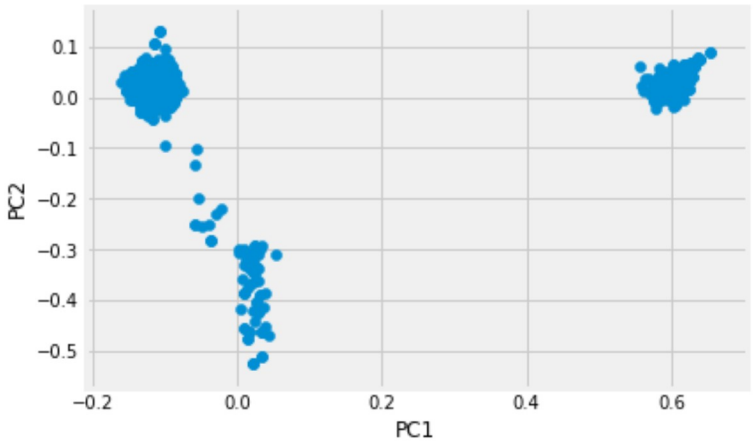
In [11]:
```
# Now let's see what we observe in our plot

from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import pandas as pd

def get_projection(X,k):
    _, _, V = np.linalg.svd(X)
    V = V.T
    return V[:, :k]

def corr_matrix_data(X):
    # demean the data
    X_demeaned = X - np.mean(X, 0)

    # get the projection matrix
    proj_matrix = get_projection(X_demeaned, 3)

    # get projected data
    X_proj = X_demeaned @ proj_matrix

    # do some k-means clustering to identify which points are in which cluster
    km = KMeans(n_clusters = 3)
    clusters = km.fit_predict(X_proj)

    # sort data based on identified clusters
    t = pd.DataFrame(X)
    t['cluster'] = clusters
    t = t.sort_values("cluster")
    t = t.drop("cluster", 1)
    plt.imshow(np.corrcoef(t.as_matrix()), "hot")
    plt.colorbar()
    plt.grid(False)

corr_matrix_data(X)
```



By reorganizing the correlation matrix a bit, we can see that our data very much contains approximately 3 clusters, so in fact our iid noise assumptions certainly do not hold. At this point it might be a good idea to start considering projecting our data in a lower dimensional space, so that we can learn more about this dependency.

In [12]:
```
# Let's project our data in 2 dimensions
X_demeaned = X - np.mean(X, 0)
proj_matrix_2d = get_projection(X_demeaned, 2)
X_proj2d = X_demeaned @ proj_matrix_2d
plt.scatter(X_proj2d[:, 0], X_proj2d[:, 1])
plt.xlabel("PC1")
plt.ylabel("PC2")
```

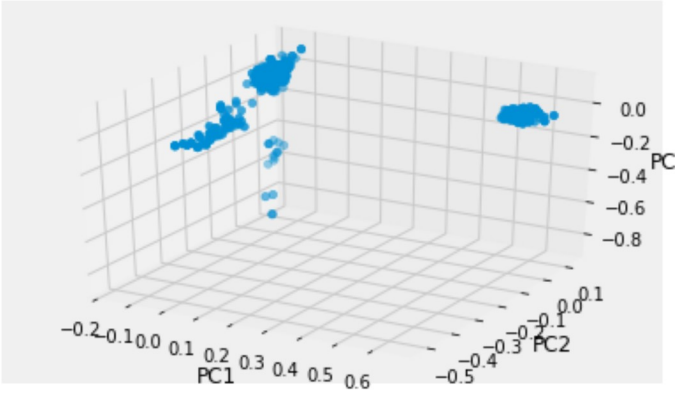
Out[12]:
```
Text(0,0.5,'PC2')
```



OR

In [13]:
```
# Now let's project on 3 dimensions
from mpl_toolkits.mplot3d import Axes3D

proj_matrix_3d = get_projection(X_demeaned, 3)
X_proj3d = X_demeaned @ proj_matrix_3d

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

xs = X_proj3d[:, 0]
ys = X_proj3d[:, 1]
zs = X_proj3d[:, 2]

ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')

ax.scatter(xs, ys, zs)
```

Out[13]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x12d5c09b0>



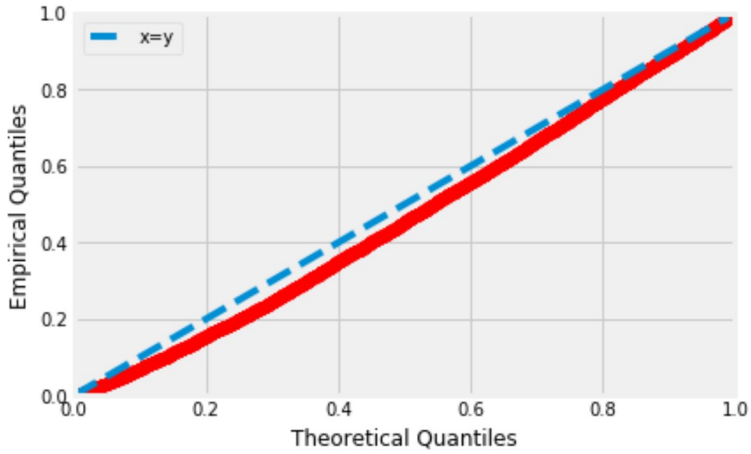
In the visualizations above, see that there is certainly some underlying structure in our data. In fact, the clusters we observe are related to the ethnicities of people in our dataset!

Part c

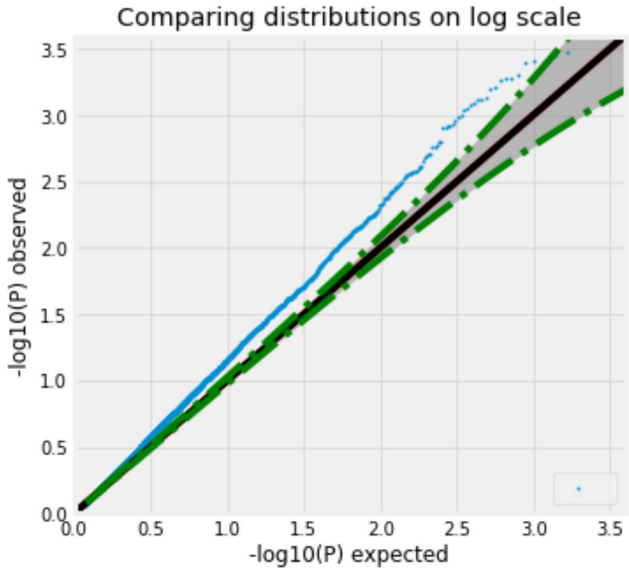
From the quantile-quantile plot in the previous part, it appears that the model is picking up on more association than theoretically expected. The reason for this is owing to the assumption of iid noise being correct. In particular, this data set contains individuals from different racial backgrounds, and also has clusters of individuals from extended families (e.g. grandparents, parents, siblings). This means that their genetics are not iid, and hence linear regression yields spurious results--all the genetic features seem to be implicated in the trait. Thus we need to modify our noise assumptions by somehow accounting for the hidden structure in the data set. The main idea is that when testing one genetic feature, all the other genetic features, jointly, are a proxy to the racial and family background. If we could include them in our model, we could correct the problem. **Ideally we would like to use all the genetic features in the linear regression model, however this is not a good idea. Why not?** Hint: There are roughly 1300 individuals and 7500 genetic variants. A written, English answer is sufficient.

So instead of using all genetic features, we will try using PCA to reduce the number of genetic features. As we saw in class, PCA on a genetic similarity matrix can capture geography, which correlates to race, quite well. So instead of adding all the genetic features, we will instead use only three features (One needs to choose this number, but we have done so for you.), X_{proj}, which are the X projected onto the top 3 principal components of X. Consequently, the updated null model is $\vec{y} = X_{proj}\overrightarrow{w_{proj}} + \vec{\epsilon}$ where $\overrightarrow{w_{proj}} \in \mathbb{R}^{3\times 1}$, while the alternative model is $\vec{y} = [\overrightarrow{x_j}, X_{proj}, \vec{1}]\vec{w} + \vec{\epsilon}$ where $\vec{w} \in \mathbb{R}^{5\times 1}$ for genetic variant j. **Plot the quantile-quantile plot from obtaining p-values with this PCA linear regression approach by running the function $pca\_corrected\_model$. How does this plot compare to the first plot? What does this tell you about this model compared to the previous model?**

In [56]:
```
"""
remember that the function pca_corrected_model returns the
empirical distribution of the p-values for the pca-corrected model
"""


pca_model_pvals = pca_corrected_model(X, y)
qqplot(pca_model_pvals)
```

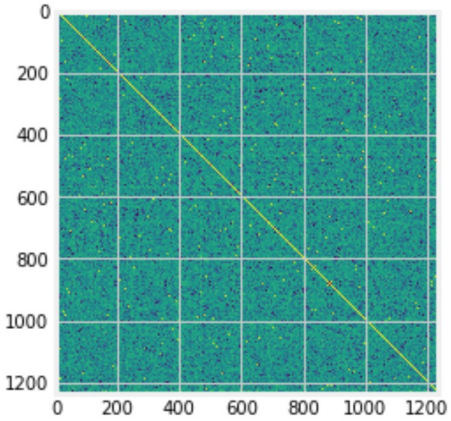



```
/Users/janeyu/anaconda/lib/python3.6/site-packages/numpy/lib/scimath.py:120: RuntimeWarning: invalid value encountered in less
  if any(isreal(x) & (x < 0)):
```



In [15]:
```
# Now, let's see if there is any patterns in our correlation matrix after we have r
emoved the
# 3 largest directions of variance identified by our PCA
X_new = X - X_proj3d  @ proj_matrix_3d.T
# change the scale a bit so that patterns are more clearly visible
corr_matrix = abs(np.corrcoef(X_new))
plt.imshow(np.log(corr_matrix+0.0001))
```

Out[15]: <matplotlib.image.AxesImage at 0x117cf9d30>



We observe that there are still some patterns that appear in our correlation matrix. That seems like evidence that we haven't captured the full extent of the relationship between our datapoints so far. Can we do better?

Part d

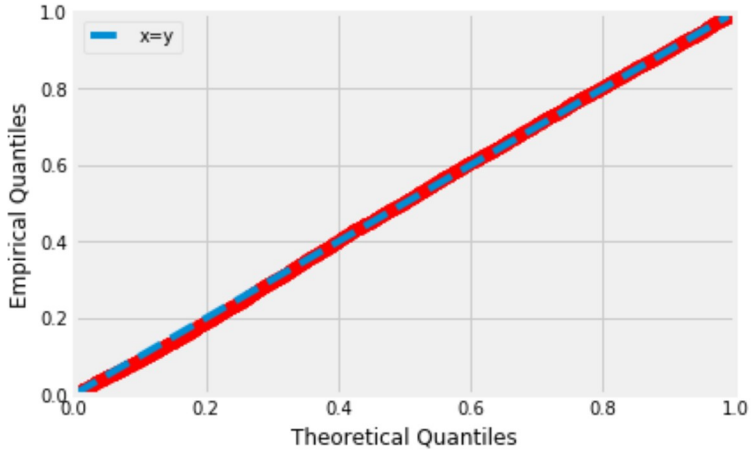
PCA got us part of the way there. However, PCA truncates the eigenspectrum; if the tail-end of that spectrum is important, as it is for family-relatedness, then it will not fully correct for our problem. So we want a method which (a) is well-behaved in terms of number of parameters that need to be estimated, and (b) includes all of the information we need. So rather than adding the projections as features, we use an modelling approach called linear mixed models which effectively adjust the iid noise in the gaussian by the pairwise genetic similarity of all the individuals. That is, we set Σ in $\vec{y} \sim N(y|[\overrightarrow{x_j}, 1]\vec{w}, I\sigma^2 + XX^\top\sigma_k^2)$.

Specifically, $\vec{y} = [\overrightarrow{x_j}, 1]\vec{w} + \vec{z} + \vec{\epsilon}$ where $\vec{z} \sim N(0, \sigma_k^2 K)$ where $K = XX^\top$, σ_k, $\vec{w} \in \mathbb{R}^{m\times 1}$, and σ are parameters we want to estimate. Notice that $\vec{y} \sim N([\overrightarrow{x_j}, 1]\vec{w}, \sigma^2 I + \sigma_k^2 K)$. Evaluation of the likelihood is thus on the order of $O(n^3)$ from the required matrix inverse and determinant of $\sigma^2 I + \sigma_k^2 K$. To test m genetic variants, the time complexity becomes $O(mn^3)$, which is extremely slow for large datasets. **Given the eigen-decomposition $K = UDU^\top$, how can we make this faster if we have to test thousands of genetic feature? Would this be computationally more efficient if you only have one genetic feature to test?**

Finally, make the quantile-quantile plot for this last model by running the function `lmm`. What can we conclude about this model relative to the other two models?

HINT: Since the manipulations needed for $\sigma^2 I + \sigma_k^2 K$ is the bottleneck here, we would like a transformation which makes this covariance of the multi-variate gaussian be a diagonal matrix.

In [57]:
```
# SOLUTION CELL
lmm_pvals = lmm()
qqplot(lmm_pvals)
```




```
/Users/janeyu/anaconda/lib/python3.6/site-packages/numpy/lib/scimath.py:120: Run
timeWarning: invalid value encountered in less
  if any(isreal(x) & (x < 0)):
```