1. (a)

I did this homework with Luning Zhao.

We work on our homework seperately, but we discuss when meeting problems.

Homework is fine.

(b) 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.

Siyao Jia

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

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

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

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

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

$= E[\langle \nabla f(w^t), w^t - w^*\rangle | i_1,\cdots,i_{t-1}]$ ↓ unbiased SGD

$= E[\langle \nabla f(w^t), w^t - w^*\rangle]$

$\Delta_{t+1} = E\|w^{t+1} - w^*\|^2$

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

$= \Delta_t - 2\alpha_t E\langle \nabla f(w^t), w^t - w^*\rangle + \alpha_t^2 E\|G_t\|^2$

$\le \Delta_t - 2\alpha_t E\langle \nabla f(w^t), w^t - w^*\rangle + \alpha_t^2 E[M_g^2 \|w^t - w^*\|^2 + B^2)$

$= (1 + \alpha_t^2 M_g^2)\Delta_t - 2\alpha_t E\langle \nabla f(w^t), w^t - w^*\rangle + \alpha_t^2 B^2$

(c) $\langle \nabla f(w^t), w^t - w^*\rangle = \langle \nabla f(w^t) - \nabla f(w^*), w^t - w^*\rangle$

$= \frac{2}{n}\langle X^T X w^t - X^T X w^*, w^t - w^*\rangle$

$= \frac{2}{n}(w^t - w^*)^T X^T X (w^t - w^*)$

$\ge \frac{2}{n}\lambda_{\min}(X^T X)\|w^t - w^*\|^2$

Let $m = \frac{2\lambda_{\min}(X^T X)}{n}$

$\Delta_{t+1} \le (1 + \alpha_t^2 M_g^2)\Delta_t - 2\alpha_t E[m\|w^t - w^*\|^2] + \alpha_t^2 B^2$

$= (1 + \alpha_t^2 M_g^2 - 2\alpha_t m)\Delta_t + \alpha_t^2 B^2$

(d) $E\|G(w)\|^2 = E\|\nabla f_i(w)\|^2$

$= E[\|2(x_i x_i^T w - x_i y_i)\|^2]$ $\quad \downarrow y_i = x_i^T w^*$

$= 4E[\|x_i x_i^T (w - w^*)\|^2]$

$= 4E[(w-w^*)^T x_i x_i^T x_i x_i^T (w-w^*)]$

$= 4E[\|x_i\|^2 \|(w-w^*)^T x_i\|^2]$

$\le 4\|x_i\|_{max}^4 E\|w - w^*\|^2 = 4\|x_i\|_{max}^4 \|w-w^*\|^2$

Let $M_g^2 = 4\|x_i\|^4$

Then $E\|G(w)\|^2 \le M_g^2 \|w-w^*\|^2$

$\Delta_t \le (1 + \alpha_t^2 M_g^2 - 2\alpha_t m)\Delta_{t-1} \le (1 + \alpha_t^2 M_g^2 - 2\alpha_t m)^t \Delta_0$

To make $1 + \alpha_t^2 M_g^2 - 2\alpha_t m$ smallest,

$$\frac{\partial(1 + \alpha_t^2 M_g^2 - 2\alpha_t m)}{\partial \alpha_t} = 2\alpha_t M_g^2 - 2m = 0 \qquad \alpha_t = \frac{m}{M_g^2}$$

$$\Delta_t \le (1 + \frac{m^2}{M_g^2} - \frac{2m^2}{M_g^2})^t \Delta_0 = (1 - \frac{m^2}{M_g^2})^t \Delta_0$$

(e) from (c), we know: $\Delta_t \le \gamma \Delta_{t-1} + \alpha^2 B^2$

$\le \gamma^2 \Delta_{t-2} + (1+\gamma)\alpha^2 B^2$

$\le \gamma^t \Delta_0 + (1 + \gamma + \cdots + \gamma^{t-1})\alpha^2 B^2$

$\le \gamma^t \Delta_0 + \frac{\alpha^2 B^2}{2m - \alpha^2 M_g^2}$

$= \gamma^t \Delta_0 + \frac{\alpha B^2}{2m - \alpha M_g^2}$

It doesn't gaurantee $w^t \to w^*$, because $\lim_{t\to\infty} \Delta_t = \frac{\alpha B^2}{2m - \alpha M_g^2} \neq 0$,

so it's not necessarily for w^t to convergo to w^*.

(4) Conclusion:

For $y = w^* x$, constant learning rate performs well, and SGD converge to w^*.

For $y = w^* x + b$, when you have constant step size,

GD converge to w^*, but SGD doesn't

when you have decreasing step size, even SGD converge to w^*.

starter_P1_SGDtheory

March 13, 2018

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pdb
        import random
        %matplotlib inline
```

1 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 [2]: 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 [22]: 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 = X.T @ X @ w_guesses[j] - X.T @ y
                    sample_gradient /= len(y)
                else:
                    # Your code, implement the gradient for SGD
                    idx = np.random.randint(len(y))
                    sample_gradient = X[[idx]].T @ X[[idx]] @ w_guesses[j] -  X[[idx]].T @

                # Your code: implement the gradient update
                # learning rate at this step is given by step_size_func(it)
                w_guesses[j] = w_guesses[j] - sample_gradient * step_size_func(it)
                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 [23]:
```
its = 5000
w_guesses, error = sgd(X, y, w, 1e-10, its, 0.0001)
```

In [24]:
```
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=True)
plt.title("Average Error vs Iterations for SGD with exact sol")
plt.legend()
plt.show()
```


```
In [25]: 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:  1064
Final average error:  2.009834128221152e-09
```

2 Part B: No solutions, constant step size

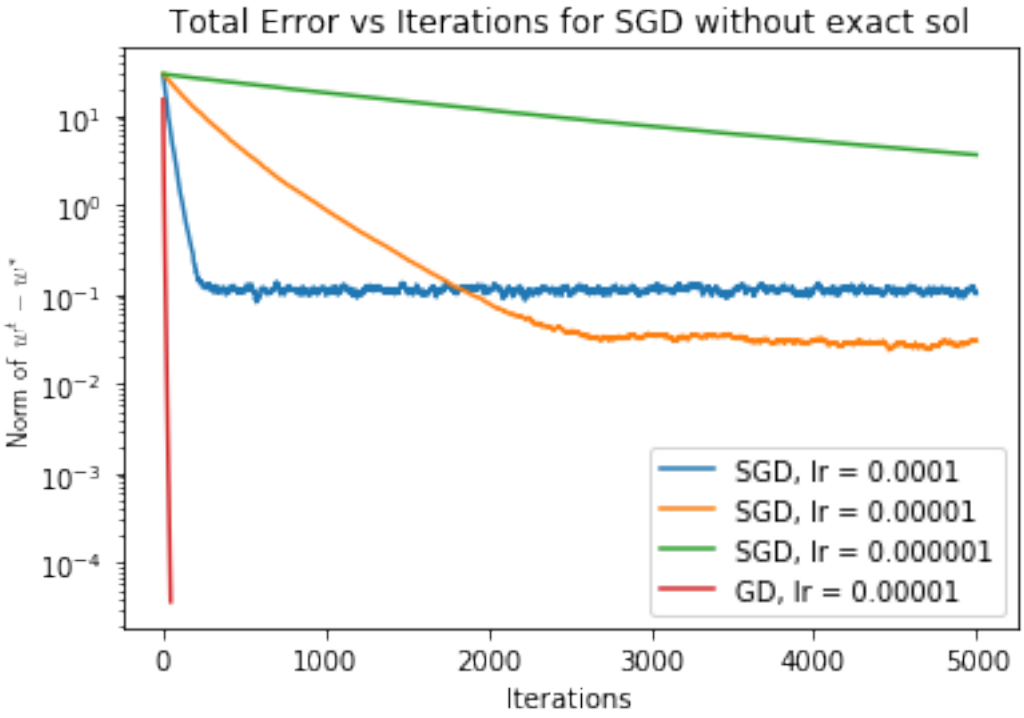
```
In [26]: y2 = y + np.random.normal(scale=5, size = y.shape)
         w=np.linalg.inv(X.T @ X) @ X.T @ y2
```

```
In [27]: 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 [28]: w_guess_gd, error_gd = sgd(X, y2, w, 1e-5, its, 0.001, True)
```

```
In [29]: 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=True)
plt.title("Total Error vs Iterations for SGD without exact sol")
plt.legend()
plt.show()
```



In [30]:
```
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:  0.10345803797006706
```

```
Required iterations, lr = 0.00001:  5000
Final average error:  0.03099382986592833
Required iterations, lr = 0.000001:  5000
Final average error:  3.685431873458876
Required iterations, GD:  47
Final average error:  3.7024294265064345e-05
```

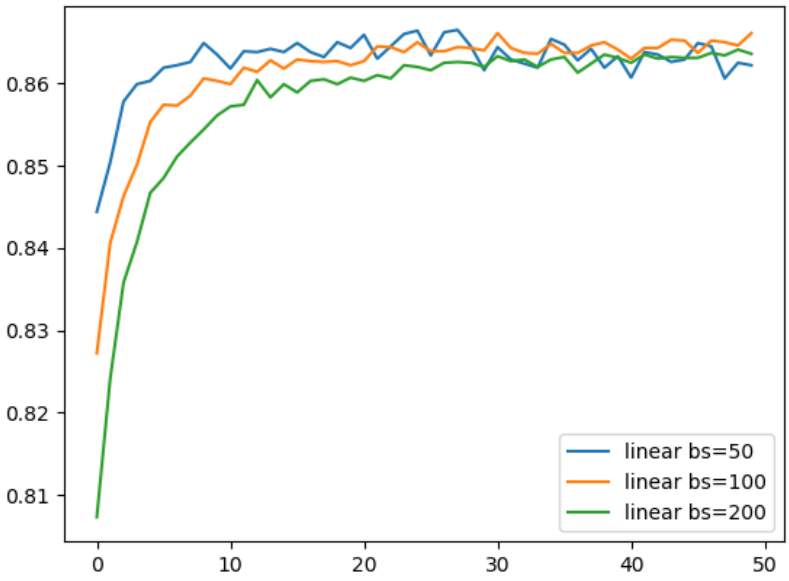
3 Part C: No solutions, decreasing step size

```
In [31]: 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 [32]: plt.semilogy([i for i in range(len(error_variable))], error_variable, label="Average er
         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 = 0.0000
         plt.semilogy([i for i in range(len(error4))], error4, label="Average error, lr = 0.0000

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


```
In [33]: print("Required iterations, variable lr: ", len(error_variable))
         average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses_variable])
         print("Average error with decreasing lr:", average_error)
```

```
Required iterations, variable lr:  5000
Average error with decreasing lr: 0.011307058669572923
```

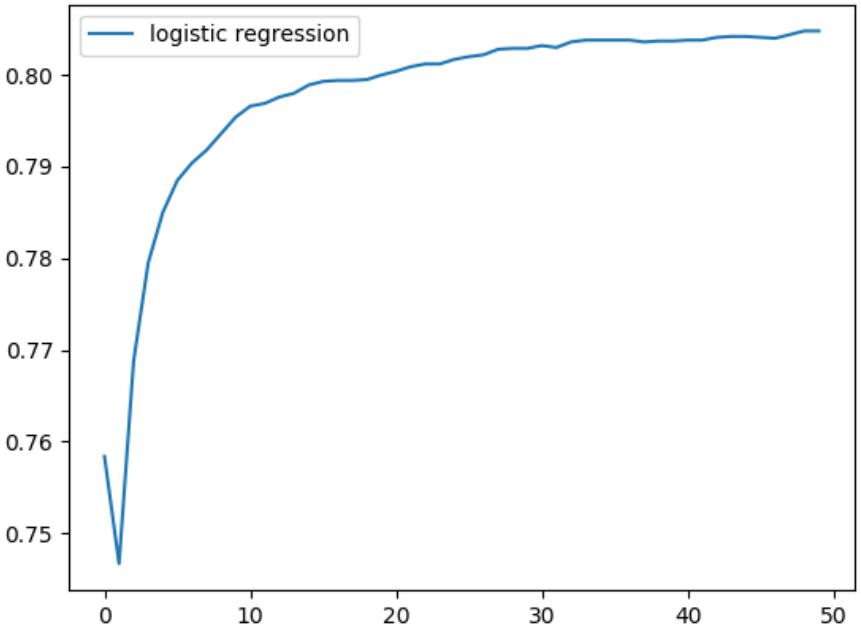
3 (a)



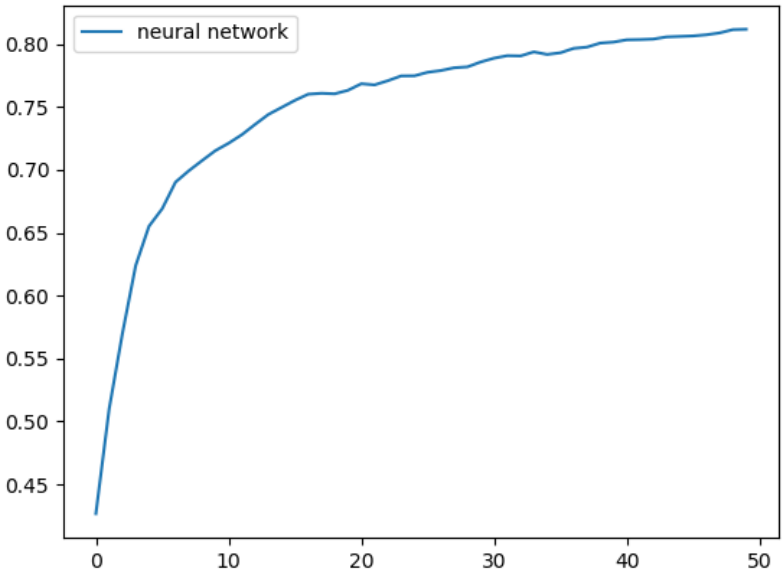
running time for batch size of 50: 63.273s
running time for batch size of 100: 47.256s
running time for batch size of 200: 42.022s

Using small batch size, the potential benefits are faster convergence, but the drawbacks are longer calculation time

(b)



(c)



(e)
1.
learning_rate = 0.01
training_epochs = 100
batch_size = 100

Epoch: 0015 cost= 57.841941198 |w-w_true|^2 = 0.017615196
TLS through SVD error: |w-w_true|^2 = 0.0003320206394634698

2. change learning rate
learning_rate = 0.05
training_epochs = 100
batch_size = 100

Epoch: 0004 cost= 56.934029833 |w-w_true|^2 = 0.051046396
TLS through SVD error: |w-w_true|^2 = 0.0003320206394634698

3. change batch_size
learning_rate = 0.01
training_epochs = 100
batch_size = 500

Epoch: 0086 cost= 248.381114960 |w-w_true|^2 = 0.014546557
TLS through SVD error: |w-w_true|^2 = 0.0003320206394634698

the solution of SGD is not sensitive to its hyper-parameters in this problem

3bc


```
import time

import matplotlib.pyplot as plt

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

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)


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 = 0
    loss = 0
    ################
    z = tf.matmul(x, W) + b
    z_max = tf.reduce_max(z)
    sm_sum = tf.reduce_sum(tf.exp(z - z_max), axis=0)
    pred = tf.exp(z - z_max)/sm_sum
    loss = (-1.0) * tf.reduce_sum(tf.log(tf.reduce_sum(tf.multiply(pred, y))))


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


def train_nn(learning_rate=0.01, training_epochs=50, batch_size=50, n_hidden=128):
    x = tf.placeholder(tf.float32, [None, 784])
    y = tf.placeholder(tf.float32, [None, 10])
```

3bc


```
    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 = 0
    loss = 0
    ################
    z1 = tf.matmul(x, W1) + b1
    z2 = tf.matmul(tf.tanh(z1), W2) + b2
    z2_max = tf.reduce_max(z2)
    sm_sum = tf.reduce_sum(tf.exp(z2 - z2_max), axis=0)
    pred = tf.exp(z2 - z2_max)/sm_sum
    loss = (-1.0) * tf.reduce_sum(tf.log(tf.reduce_sum(tf.multiply(pred, y))))


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


def main():
    #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" % (time.time() - time_start))

    #    plt.plot(acc_linear, label="linear bs=%d" % batch_size)
    #    plt.legend()
    #    plt.savefig('linear_%d.png'%batch_size, format='png')

    #acc_logistic = train_logistic()
    #plt.plot(acc_logistic, label="logistic regression")
    #plt.legend()
    #plt.show()

    acc_nn = train_nn()
    plt.plot(acc_nn, label="neural network")
    plt.legend()
    plt.show()


if __name__ == "__main__":
    tf.set_random_seed(0)
    main()
```

3e


```
import numpy as np

import tensorflow as tf

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 = 0.01
training_epochs = 100
batch_size = 100


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
    cost = 0
    ################
    cost = 0.5*tf.log(tf.square(tf.norm(w))+1) + 0.5/(tf.square(tf.norm(w))+1)*tf.sq
uare(tf.norm(y - tf.matmul(x, w)))

    # 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.AdamOptimizer(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))


if __name__ == "__main__":
    tf.set_random_seed(0)
    np.random.seed(0)
    main()
```

3(d) $y_i = w^T x_i - w^T z_x + z_y$

$z_x \sim N(0,1) \quad w^T z_x = \sum_{i=1}^{d} w_i z_{xi} \sim N(0, \Sigma w_i^2) \sim N(0, \|w\|^2)$

$z_y \sim N(0,1)$

$\Rightarrow y_i \sim N(w^T x_i, \ 1 + \|w\|^2)$

$$\sum_{i=1}^{n} \log P(y_i | x_i) = \sum_{i=1}^{n} \log \frac{}{\sqrt{2\pi(1+\|w\|^2)}} e^{-\frac{(y_i - w^T x_i)^2}{2(1+\|w\|^2)}}$$

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

$$= -\frac{n}{2} \log(1+\|w\|^2) - \frac{1}{2(1+\|w\|^2)} \sum_{i=1}^{n} (y_i - w^T x_i)^2 + C$$

4(d) $\sigma^2 I + \sigma_k^2 k = \sigma^2 I + \sigma_k^2 U D U^T$

$= U(\sigma^2 I + \sigma_k^2 D) U^T$

$P(y) \propto e^{-[y-\hat{y}]^T [U(\sigma^2 I + \sigma_k^2 D)U^T]^{-1} [y-\hat{y}]}$ where $\hat{y} = [X \ 1] w$

$= e^{-[y-\hat{y}]^T U (\sigma^2 I + \sigma_k^2 D)^{-1} U^T [y-\hat{y}]}$

Let $\tilde{y} = U^T [y - \hat{y}]$

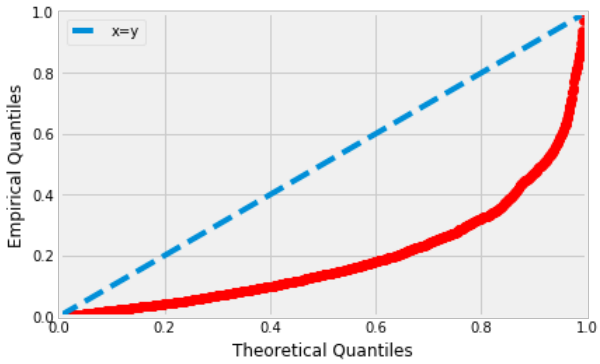
Then $\tilde{y} \sim N(0, \sigma^2 I + \sigma_k^2 D)$

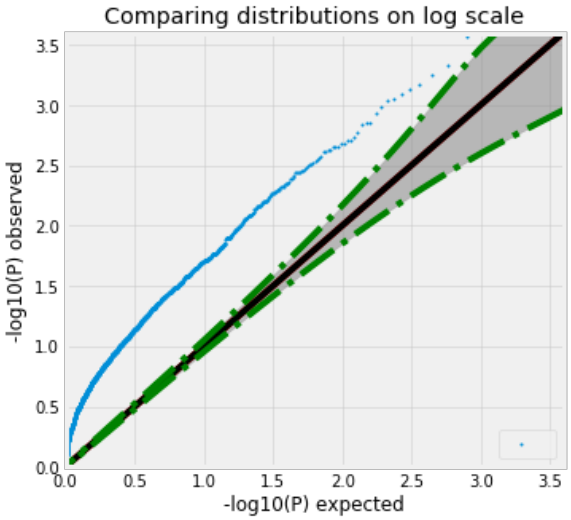
This will make computation faster.

No, if we only have one feature, this will not be more efficient.

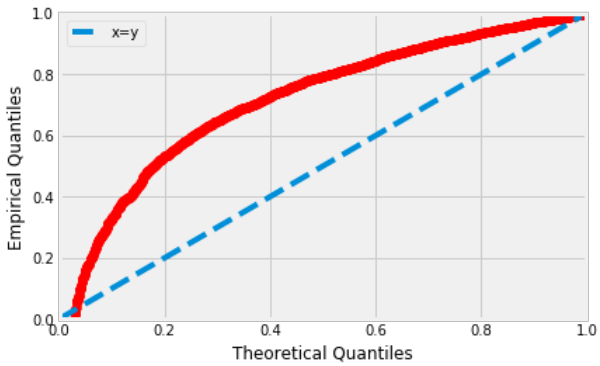
4 (a)

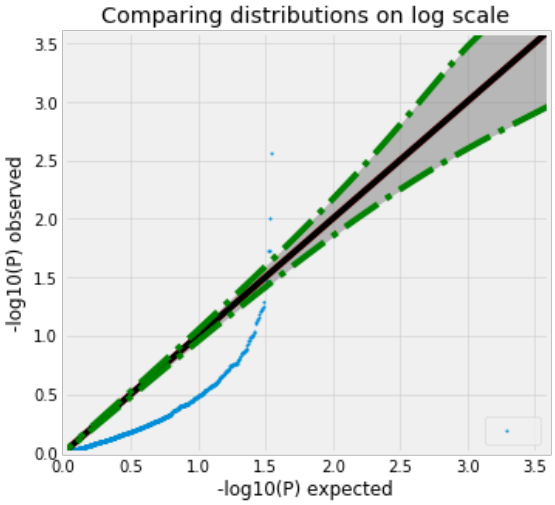
Skewleft



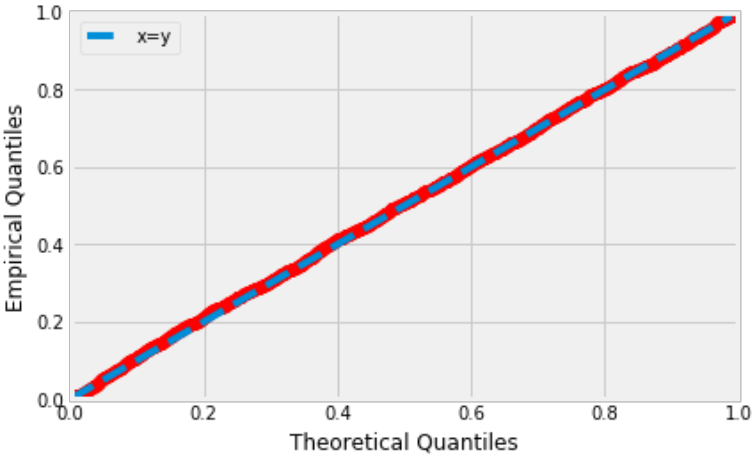


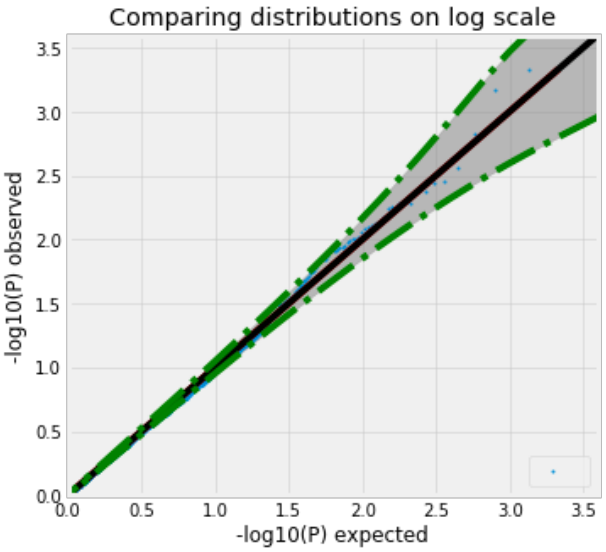
Skew right





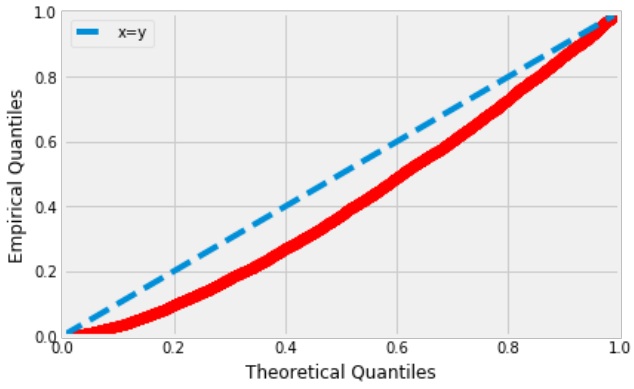
uniform

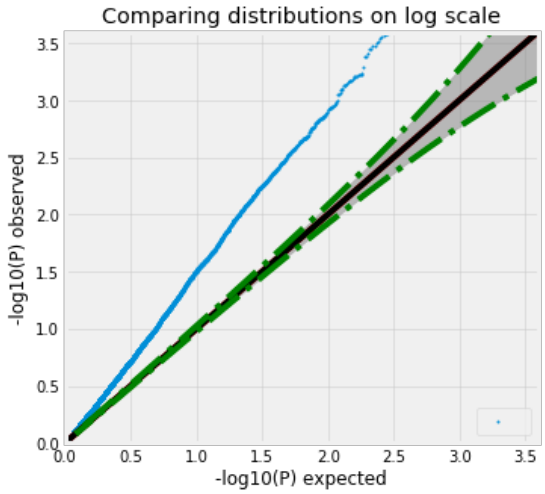




I found that for uniform distribution, then the empirical quantiles lies on top of theoretical quantiles, meaning that. For Left-skewed distribution or right-skewed distortion, the empirical quantiles are different with theoretical quantiles.
We should see the observed line gets more and more closer to theoretical line if our empirical distribution looks more and more similar to a Unif[0, 1] .

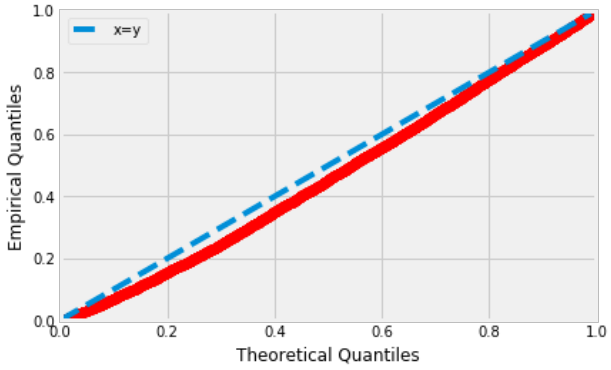
(b)

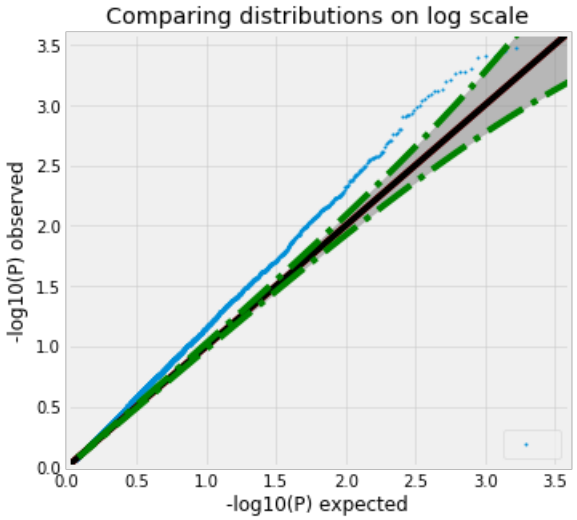




From the plot, we see the difference between empirical and theoretical quantiles, so linear regression is not suitable for our problem.

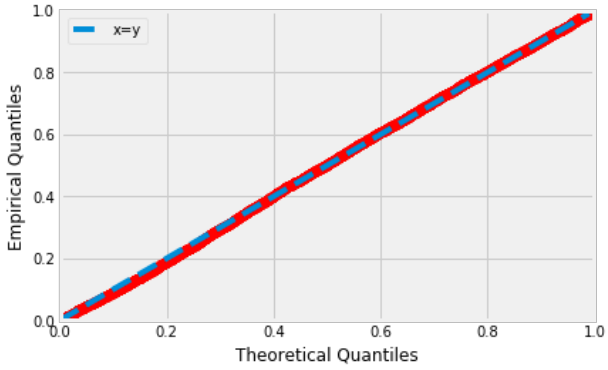
(c)
Using all genetic features will be too many free parameters, it's computationally expensive. More importantly, we'll have more variants and data, so we will overfit our data.

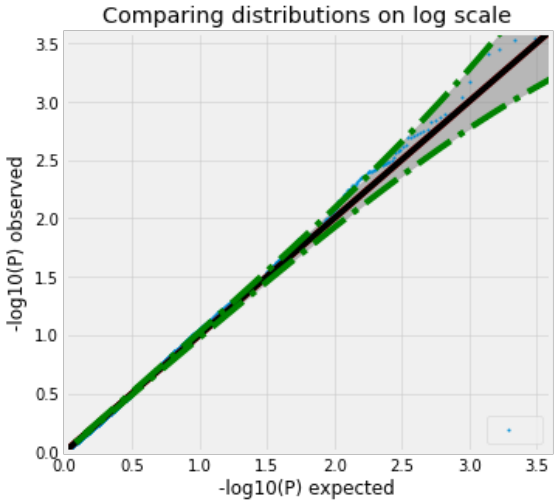




This plot is the quantiles agrees better, so PCA is a better model than pure linear regression.

(d)





From the plot, we can see this model is the best among the three models.

5. Your own question.

Q: Suppose you want to solve $f(x)=0$ with iterative methods.

say you start from $x=x_0$, then $y = f(x_0)(x-x_0)+f(x_0)$

$$\Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Compute the convergence rate.

A: Suppose at some x_i, $f(x_i)=0$

and x_n is an estimate of x_i, so $|x_i - x_n| = \delta \ll 1$

$$f(x_i) = f(x_n+\delta) = f(x_n) + f'(x_n)(x_i - x_n) + \frac{f''(x_j)}{2}(x_i - x_n)^2 \quad ①$$

where $x_n < x_j < x_i$.

$$f(x_n) = f'(x_n)(x_n - x_{n+1}) \quad ②$$

Combine ① and ② $\Rightarrow$ $f'(x_n)\underline{(x_i - x_{n+1})} = \frac{f''(x_j)}{2}\underline{(x_i - x_n)^2}$.

So we can see it converges quadratically.