Question 1. Getting Started

You may typeset your homework in latex or submit neatly handwritten and scanned solutions. Please make sure to start each question on a new page, as grading (with Gradescope) is much easier that way! Deliverables:

1. Submit a PDF of your writeup to assignment on Gradescope, "HW[n] Write-Up"
2. Submit all code needed to reproduce your results, "HW[n] Code".
3. Submit your test set evaluation results, "HW[n] Test Set".

After you've submitted your homework, be sure to watch out for the self-grade form.

(a) 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 with Weiran Liu and Katherine Li. I feel really sick right now but I really want to crack the homework ASAP. I'm glad to see a entire question becomes the BONUS not only a few parts. But I will do it.

(b) Please copy the following statement and sign next to it. We just want to make it *extra* clear so that no one inadvertently cheats.

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.

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.

Signature: 

Question 2. SGD on OLS

In this problem, we carefully walk through the key ingredients of a proof for SGD convergence for the specific example of a loss function which we are very familiar with: Ordinary Least Squares with positive definite matrix $\mathbf{X}^\top\mathbf{X}$. In particular we show that in this case, even though gradient descent converges to the optimal solution of OLS, with small enough constant stepsize, SGD is not guaranteed to do so! We then show that in contrast, when applying SGD with decaying step, as outlined in lecture, it does converge to the same optimum as gradient descent.

This phenomenon that constant stepsize gets stuck and decaying stepsizes are the way out is analogous to common observations in practical machine learning problems, where decreasing the learning rate using heuristic learning rate schedules helps to decrease the training error.

Recall that the ordinary least squares problem can be written as:

$$\min_{\vec{w}} f(\vec{w}) = \min_{\vec{w}} \frac{1}{n}||\mathbf{X}\vec{w} - \vec{y}||_2^2 = \min_{\vec{w}} \frac{1}{n}\sum_{i=1}^{n}(\vec{x}_i^\top \vec{w} - \vec{y}_i)^2 = \min_{\vec{w}} \frac{1}{n}\sum_{i=1}^{n} f_i(\vec{w})$$

where $f_i(\vec{w}) := (\vec{x}_i^\top \vec{w} - \vec{y}_i)^2$. Here $\vec{x}_i^\top$ is the ith row of matrix $\mathbf{X}$ and f_i is the loss of the training example i. We implement SGD as follows:

$$\vec{w}^{t+1} = \vec{w}^t - \alpha_t \nabla f_{i_t}(\vec{w}^t).$$

where i_t is uniformly sampled from all samples $\{1, 2, \cdots, n\}$ (and is independently drawn for each iteration t). Let's define the short hand $G_t = \nabla f_{i_t}(\vec{w}^t)$ to represent the random gradient at step t. We are interested in how $\vec{w}^t$ approaches the optimal solution $\vec{w}^* := \arg\min_{\vec{w}\in\mathbb{R}^d} f(\vec{w})$. One way to characterize this is to monitor the squared distance of the iterate to the optimum, i.e. $||\vec{w}^t - \vec{w}^*||_2^2$. Throughout, we will assume that the following bound holds for the squared norm of the stochastic gradient:

$$\mathbb{E}||G(\vec{w})||_2^2 \leq M_g^2||\vec{w} - \vec{w}^*||_2^2 + B^2. \tag{1}$$

where M_g and B are constants dependent on the model and loss function f We will find concrete values of them later in this problem for specific examples. (Notice that in lecture we assumed M_g to be zero, which is too restrictive even for the most basic least squares loss.)

Problem outline: Parts (a)-(c) help to derive a recursive formula of the form $\mathbb{E}||\vec{w}^{t+1} - \vec{w}^*||_2^2 \leq \gamma\mathbb{E}||\vec{w}^t - \vec{w}^*||_2^2 + \tilde{\gamma}$. In part (d) we show that we achieve linear (also called geometric) convergence of $\vec{w}^t$ to the optimum $\vec{w}^*$ for SGD with constant stepsize when $f(\vec{w}^*) = 0$. That means we can write $\mathbb{E}||\vec{w}^t - \vec{w}^*||_2^2 \leq \gamma^t||\vec{w}^0 - \vec{w}^*||_2^2$ for some $\gamma < 1$. In part (e) we explore what happens with the iterates of SGD with constant stepsize if instead $f(\vec{w}^*) > 0$. In part (f) we visualize how the convergence bounds translate to actual training error decrease for OLS using SGD and GD.

The bonus parts (g)-(h) are for those students who "love math" according to the midterm preliminary questions, where you are asked to prove that SGD with decaying stepsize converges as $1/t$ for "nice" functions using induction.

(a) **Show the following relation between the $t+1$-step error and the t-step error:** $||\vec{w}^{t+1} - \vec{w}^*||_2^2 = ||\vec{w}^t - \vec{w}^*||_2^2 - 2\alpha_t\langle G_t, \vec{w}^t - \vec{w}^*\rangle + \alpha_t^2||G_t||_2^2$

$$
\begin{aligned}
&\|\vec{w}^{t+1} - \vec{w}^*\|_2^2 \\
=&\|\vec{w}^t - \alpha_t \nabla f_{it}(\vec{w}^t) - \vec{w}^*|_2^2 \\
=&\|\vec{w}^t - \vec{w}^* - \alpha_t G^t\|_2^2 \\
=&(\vec{w}^t - \vec{w}^*)^\top(\vec{w}^i - \vec{w}^*) - \alpha_t(\vec{w}^t - \vec{w}^*)^\top G^t - \alpha_t(G^t)^\top(\vec{w}^t - \vec{w}^*) + (\alpha_t G^t)^\top(\alpha_t G^t) \\
=&\|\vec{w}^t - \vec{w}^*\|_2^2 - 2\alpha_t < G^t, \vec{w}^t - \vec{w}^* > + \alpha_t^2\|G^t\|_2^2
\end{aligned}
$$

(b) *Here we prove the key relation which underlies the success of stochastic gradient methods. This is where we use that stochastic gradients are unbiased!* Since we have stochastic gradients, we want to make guarantees in terms of expectations. For notational convenience let us define the short hand $\Delta_t := \mathbb{E}||\vec{w}^t - \vec{w}^*||_2^2$, the *average squared error*, where the expectation is taken over all the random indeces drawn by the stochastic gradient method up to time t. We want to show a clean relation between Δ_t and Δ_{t+1} when the stochastic gradient satisfies (1).

Remind yourself which random indeces $\vec{w}^t$ depends on and use the unbiasedness of the stochastic gradient to **show that**

$$\mathbb{E}[\langle G(\vec{w}^t), \vec{w}^t - \vec{w}^\star \rangle] = \mathbb{E}[\langle \nabla f(\vec{w}^t), \vec{w}^t - \vec{w}^\star \rangle]. \tag{2}$$

Now use equality (2) and (a) to prove that

$$\Delta_{t+1} \leq (1 + \alpha_t^2 M_g^2)\Delta_t - 2\alpha_t \mathbb{E}\langle \nabla f(\vec{w}^t), \vec{w}^t - \vec{w}^* \rangle + \alpha_t^2 B^2. \tag{3}$$

Hint 1: You may take the law of iterated expectation (also called tower property) as given, i.e.

$$\mathbb{E}[\langle G(\vec{w}^t), \vec{w}^t - \vec{w}^\star \rangle] = \mathbb{E}_{i_1,\ldots,i_{t-1}}[\mathbb{E}_{i_t}[\langle G(\vec{w}^t), \vec{w}^t - \vec{w}^\star \rangle | i_1, \ldots, i_{t-1}]].$$

See Discussion for the derivation.

Hint 2: Unbiased stochastic gradient means that $\mathbb{E}_{i_t} \nabla f_{i_t}(\vec{w}) = \nabla f(\vec{w})$ for any $\vec{w}$ independent of the random index, where the expectation is with respect to the random index i_t at time t. Refer to the Discussion for brief intro on conditional expectations.

We know that randomness of SGD comes from random sampling. That is, we only pick a subset of sample points and use that to update our entire weight vector $\vec{w}^t$. If we assume all data points are drawn iid, we have:

Thus computing $\nabla_w L$ with any subset of samples $\{i\}$ gives an unbiased estimate of $\nabla_w L$:

$$E_{\{i\}}[\nabla_w L(w, x_i, y_i) - \nabla_w L] = 0$$

This is a direct photo copy of the lecture slide.

$$\begin{aligned}
&\mathbb{E}\left[< G(\vec{w}^t), \vec{w}^t - \vec{w}^* >\right] \\
=&\mathbb{E}_{i_1,\ldots,i_{t-1}}[\mathbb{E}_{i_t}[\langle G(\vec{w}^t), \vec{w}^t - \vec{w}^\star \rangle | i_1, \ldots, i_{t-1}]] \\
=&\mathbb{E}_{i_1,\ldots,i_{t-1}}[\langle \nabla f(\vec{w}^t), \vec{w}^t - \vec{w}^\star \rangle] \\
=&\mathbb{E}\left[< \nabla f(\vec{w}^t), \vec{w}^t - \vec{w}^* >\right]
\end{aligned}$$

The above is true because inner product is bilinear so we can push the expectation into the inner product; the randomness of $\nabla f(\vec{w}^t)$ comes from step 1 to $t-1$ not at step t.

Now we use the above equality to prove:

$$\begin{aligned}
&\Delta_{t+1} \\
=&\mathbb{E}\left[\|\vec{w}^{t+1} - \vec{w}^*\|_2^2\right]
\end{aligned}$$

$$
\begin{aligned}
=&\mathbb{E}\left[\|\vec{w}^t-\vec{w}^*\|_2^2-2\alpha_t<G^t,\vec{w}^t-\vec{w}^*>+\alpha_t^2\|G^t\|_2^2\right]\\
=&\mathbb{E}\left[\|\vec{w}^t-\vec{w}^*\|_2^2\right]-2\mathbb{E}\left[\alpha_t<G^t,\vec{w}^t-\vec{w}^*>\right]+\mathbb{E}\left[\alpha_t^2\|G^t\|_2^2\right]\\
=&\Delta_t-2\alpha_t\mathbb{E}\left[<G^t,\vec{w}^t-\vec{w}^*>\right]+\alpha_t^2\mathbb{E}\left[\|G^t\|_2^2\right]\\
=&\Delta_t-2\alpha_t\mathbb{E}\left[<\nabla f(\vec{w}^t),\vec{w}^t-\vec{w}^*>\right]+\alpha_t^2\mathbb{E}\left[\|G^t\|_2^2\right]\\
\leq&\Delta_t-2\alpha_t\mathbb{E}\left[<\nabla f(\vec{w}^t),\vec{w}^t-\vec{w}^*>\right]+\alpha_t^2(M_g^2\mathbb{E}\left[\|\vec{w}^t-\vec{w}^*\|_2^2\right]+B^2)\\
=&(1+\alpha_t^2M_g^2)\Delta_t-2\alpha_t\mathbb{E}\left[<\nabla f(\vec{w}^t),\vec{w}^t-\vec{w}^*>\right]+\alpha_t^2B^2
\end{aligned}
$$

(c) *In this step we see that we need for our analysis that our loss function is "nice" (strongly convex to be precise), which holds for OLS when* $\lambda_{min}(\mathbf{X}^\top\mathbf{X}) > 0$. *That is, all the guarantees for SGD we discuss in this problem only hold for such "nice" functions. In general, at least convexity is needed for any convergence proof.* **Using inequality** (3)**, now show that** $\Delta_{t+1} \leq (1+\alpha_t^2 M_g^2 - 2\alpha_t m)\Delta_t + \alpha_t^2 B^2$**, where we assume that the minimum eigenvalue of matrix** $\mathbf{X}^\top\mathbf{X}$ **denoted by** m **is positive, i.e.** $m := \frac{2}{n}\lambda_{min}(\mathbf{X}^\top\mathbf{X}) > 0$**.**

Hint: Use the fact that $\nabla f(\vec{w}^*) = \vec{0}$, and hence

$$\langle \nabla f(\vec{w}^t), \vec{w} - \vec{w}^* \rangle = \langle \nabla f(\vec{w}^t) - \nabla f(\vec{w}^*), \vec{w} - \vec{w}^* \rangle.$$

Recall the derivative of OLS optimization function:

$$\nabla f(\vec{w}) = \frac{2}{n}(\mathbf{X}^\top\mathbf{X}\vec{w} - \mathbf{X}^\top\vec{y})$$

$$\begin{aligned}
&\mathbb{E}\left[< \nabla f(\vec{w}^t) - \nabla f(\vec{w}^*), \vec{w}^t - \vec{w}^* >\right] \\
=&\mathbb{E}\left[(\nabla f(\vec{w}^t) - \nabla f(\vec{w}^*))^\top(\vec{w}^t - \vec{w}^*)\right] \\
=&\mathbb{E}\left[\frac{2}{n}(\mathbf{X}^\top\mathbf{X}\vec{w}^t - \mathbf{X}^\top\mathbf{X}\vec{w}^*)^\top(\vec{w}^t - \vec{w}^*)\right] \\
=&\mathbb{E}\left[\frac{2}{n}(\vec{w}^t - \vec{w}^*)^\top\mathbf{X}^\top\mathbf{X}(\vec{w}^t - \vec{w}^*)\right] \\
\geq&\frac{2}{n}\mathbb{E}\left[\lambda_{min}(\mathbf{X}^\top\mathbf{X})\|\vec{w}^t - \vec{w}^*\|_2^2\right]
\end{aligned}$$

The last step is true due to the lower bound of Rayleigh quotient where we have:

$$\lambda_{min}(\mathbf{X}^\top\mathbf{X}) \leq \frac{\vec{w}^\top\mathbf{X}^\top\mathbf{X}\vec{w}}{\|\vec{w}\|} \leq \lambda_{max}(\mathbf{X}^\top\mathbf{X})$$

$$\begin{aligned}
&\Delta_{t+1} \\
\leq&(1+\alpha_t^2 M_g^2)\Delta_t - 2\alpha_t\mathbb{E}\left[< \nabla f(\vec{w}^t), \vec{w}^t - \vec{w}^* >\right] + \alpha_t^2 B^2 \\
\leq&(1+\alpha_t^2 M_g^2)\Delta_t - 2\alpha_t\mathbb{E}\left[< \nabla f(\vec{w}^t) - \nabla f(\vec{w^*}^t), \vec{w}^t - \vec{w}^* >\right] + \alpha_t^2 B^2 \\
\leq&(1+\alpha_t^2 M_g^2)\Delta_t - 2\alpha_t\frac{2}{n}\mathbb{E}\left[\lambda_{min}(\mathbf{X}^\top\mathbf{X})\|\vec{w}^t - \vec{w}^*\|_2^2\right] + \alpha_t^2 B^2 \\
\leq&(1+\alpha_t^2 M_g^2 - 2\alpha_t m)\Delta_t + \alpha_t^2 B^2
\end{aligned}$$

(d) We now examine how close we can get to the optimal solution $\vec{w}^*$ using SGD with **constant stepsize**, i.e. $\alpha_t = \alpha$ for all t in two different scenarios. First, for this question we assume that $\mathbf{X}\vec{w}^* = \vec{y}$, or in other words, that the minimum loss is 0. **Show that inequality** (1) **holds with** $B = 0$ **and** $M_g^2 = 4\max_i ||\vec{x}_i||_2^4$ **in this case. Find the optimum learning rate and show that** $\Delta_t \leq (1 - \frac{m^2}{M_g^2})^t \Delta_0$**.** This means that with $B = 0$ we have linear (geometric) convergence!

Hint: You may want to use that for any matrix $\vec{A}$ and vector $\vec{w}$ it holds that $\|\vec{A}\vec{w}\|_2^2 \leq \lambda_{max}(\vec{A}^\top \vec{A})\|\vec{w}\|_2^2$ and the same bound applies when taking the expectation on both sides.

In the derivation below, X denotes the random feature vector. (Or you might view it as one data point)

$$
\begin{aligned}
&\mathbb{E}\left[\|G(\vec{w})\|_2^2\right] \\
=&\mathbb{E}\left[\|2XX^\top\vec{w} - 2X\vec{y}\|_2^2\right] \\
=&4\mathbb{E}\left[\|XX^\top\vec{w} - XX^\top\vec{w}^*\|_2^2\right] \\
=&4\mathbb{E}\left[\|XX^\top(\vec{w} - \vec{w}^*)\|_2^2\right] \\
\leq&4\lambda_{max}(XX^\top)^2\|\vec{w} - \vec{w}^*\|_2^2 \\
\leq&4(X^\top X)^2\|\vec{w} - \vec{w}^*\|_2^2 \\
=&4\|X\|_2^4\|\vec{w} - \vec{w}^*\|_2^2
\end{aligned}
$$

The third to the last step to the second to the last step is true because in HW0 we had:

$$
\lambda(\vec{q}\vec{p}^\top) = \begin{cases} 0 \\ \vec{p}^\top\vec{q} \end{cases}
$$

In this case, $\vec{p} = \vec{q}$ so $\vec{p}^\top\vec{q} = \|\vec{p}\|^2$. We only need to take the maximum observed $\max_i ||\vec{x}_i||_2^4$ to make sure the above inequality holds.

$$
\mathbb{E}\left[\|G(\vec{w})\|_2^2\right] \leq 4\max_i ||\vec{x}_i||_2^4\|\vec{w} - \vec{w}^*\|_2^2
$$

Now let's look for the optimum learning rate:

$$
\begin{aligned}
&\Delta_t \\
\leq&(1 + \alpha^2 M_g^2 - 2\alpha m)\Delta_{t-1} + \alpha_t^2 B^2 \\
=&(1 + \alpha^2 M_g^2 - 2\alpha m))\Delta_{t-1} \\
\leq&(1 + M_g^2\alpha^2 - 2m\alpha)^t\Delta_0
\end{aligned}
$$

The optimal learning rate is achieved when $1 + M_g^2\alpha^2 - 2m\alpha$ is the smallest. This is a quadratic function so we have:

$$
\alpha = \frac{-2m}{-2M_g^2} = \frac{m}{M_g^2}
$$

$$
\begin{aligned}
&\Delta_t \\
\leq &(1 + M_g^2(\frac{m}{M_g^2})^2 - 2m\frac{m}{M_g^2})^t\Delta_0 \\
= &(1 + \frac{m^2}{M_g^2} - \frac{2m^2}{M_g^2})^t\Delta_0 \\
= &(1 - \frac{m^2}{M_g^2})^t\Delta_0
\end{aligned}
$$

(e) Instead of assuming that there exists a $\vec{w}^*$ that satisfies $\mathbf{X}\vec{w}^* = \vec{y}$ exactly, we now consider the case where $f(\vec{w}^*) = \frac{1}{2}\|\mathbf{X}\vec{w}^* - y\|^2 > 0$. One can show that in this case $\mathbb{E}||G(\vec{w})||_2^2 \leq M_g^2||\vec{w} - \vec{w}^*||_2^2 + B^2$ holds for some M_g and B, where $M_g \neq 0$ and $B \neq 0$. Suppose that you are given the constants M_g and B and the stepsize is still constant, i.e. $\alpha_t = \alpha$ for all t.

We define the short hand notation $\gamma := 1 + \alpha^2 M_g^2 - 2\alpha m$ and assume α is large enough such that $\gamma < 1$. **Using part (c), prove that** $\Delta_t \leq \gamma^t \Delta_0 + \frac{\alpha B^2}{2m - \alpha M_g^2}$**.** We now want to interpret this bound. **Does it guarantee convergence** $\vec{w}^t \to \vec{w}^\star$ **when** t **goes to infinity? Explain.**

Hint: You may find the following inequality helpful: $\sum_{t=1}^{n} \gamma^t \leq \frac{1}{1-\gamma}$ for $\gamma < 1$.

In part (c), we have:

$$\begin{aligned}
&\Delta_t \\
\leq&(1 + \alpha^2 M_g^2 - 2\alpha m)\Delta_{t-1} + \alpha^2 B^2 \\
\leq&\gamma \Delta_{t-1} + \alpha^2 B^2
\end{aligned}$$

This is a arithmetico-geometric sequence. Let's do the following transformation:

$$\begin{aligned}
\Delta_t \leq&\gamma \Delta_{t-1} + \alpha^2 B^2 \\
\Delta_t + \frac{\alpha^2 B^2}{\gamma - 1} \leq&\gamma(\Delta_{t-1} + \frac{\alpha^2 B^2}{\gamma - 1}) \\
\Delta_t + \frac{\alpha^2 B^2}{\gamma - 1} \leq&\gamma^t(\Delta_0 + \frac{\alpha^2 B^2}{\gamma - 1}) \\
\Delta_t \leq&\gamma^t \Delta_0 + \gamma^t \frac{\alpha^2 B^2}{\gamma - 1} - \frac{\alpha^2 B^2}{\gamma - 1} \\
\Delta_t \leq&\gamma^t \Delta_0 - \frac{\alpha^2 B^2}{\gamma - 1} \\
\Delta_t \leq&\gamma^t \Delta_0 - \frac{\alpha^2 B^2}{\alpha^2 M_g^2 - 2\alpha m} \\
\Delta_t \leq&\gamma^t \Delta_0 - \frac{\alpha B^2}{\alpha M_g^2 - 2m} \\
\Delta_t \leq&\gamma^t \Delta_0 + \frac{\alpha B^2}{2m - \alpha M_g^2}
\end{aligned}$$

Because $\gamma < 1$, we have $2m - \alpha M_g^2 > 0$, it doesn't guarantee that $\lim_{t \to \infty} \Delta_t = 0$ and we can only prove it is smaller than a constant but not a infinity small number.

(f) We now want use simulations to compare the first order methods for solving OLS for the cases in (d) and (e). In particular, we want to plot the estimation error $||\vec{w}^t - \vec{w}^*||_2$ against the gradient descent step in the following 3 scenarios:

- When there exists an exact solution $\mathbf{X}\vec{w}^* = \vec{y}$, plot the errors when you are using SGD and a constant learning rate.
- When there does not exist an exact solution $\mathbf{X}\vec{w}^* = \vec{y}$, plot the errors when you are using SGD and a constant learning rate. Also plot the errors for GD and the same constant learning rate.
- When there does not exist an exact solution $\mathbf{X}\vec{w}^* = \vec{y}$, plot the errors when you are using SGD and a decaying learning rate.

Using the attached starter code, **implement the gradient updates** in the function `sgd()` , while all the plotting functions are already there. **Show the 3 plots you obtain using the starter code. Report the average squared error computed in the starter code. What's your conclusion?**

We can see on the first graph the error decreases monotonically and get below 10^{-8}. It means SGD with exact solution can find the optimal point.

On the second graph, we can see all SGD curves plateau after iterations except for the one with the smallest learning rate. It has a much larger error than the optimal point found by GD. It means SGD cannot find the optimal point without exact solution.

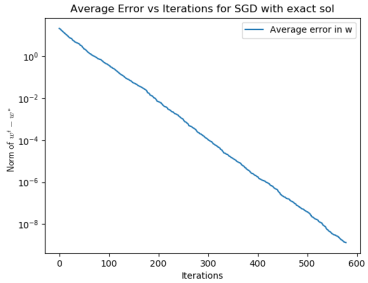
The last graph shows that the blue curve with decreasing learning rate for SGD can potentially find the optimal point because the error is still decreasing.

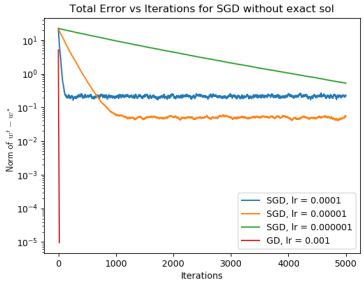
Conclusion 1: SGD with a constant learning rate is able to find the optimal solution if there exists an exact solution$\mathbf{X}\vec{w}^* = \vec{y}$.

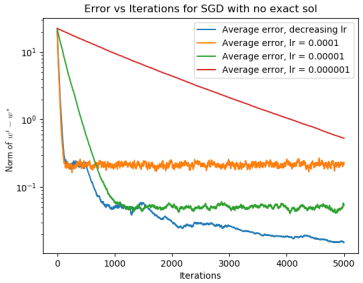
Conclusion 2: SGD with a constant learning rate cannot find the optimal solution if there is not an exact solution$\mathbf{X}\vec{w}^* = \vec{y}$.

Conclusion 3: SGD with a decaying learning rate is able to find the optimal solution even though there is no exact solution$\mathbf{X}\vec{w}^* = \vec{y}$.

Below is the SGD with one sample for one iteration. The other implementation for a more fair comparison using mini batch is attached after this. We can see more clearly for the mini batch result that without exact solution, SGD won't converge to the optimal point.







Required iterations: 581

Final average error:

$$[1.3840365847668996e-09]$$

Required iterations, lr = 0.0001: 5000

Final average error:

$$[0.2241931362287866]$$

Required iterations, lr = 0.00001: 5000
Final average error:

$$[0.05351301821666841]$$

Required iterations, lr = 0.000001: 5000
Final average error:

$$[0.5315176256579613]$$

Required iterations, GD: 18
Final average error:

$$[9.585511545807165e-06]$$

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

$$[0.01506567101702303]$$

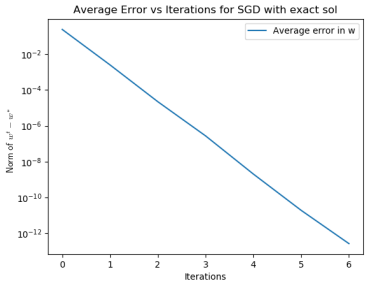
```
import numpy as np
import matplotlib.pyplot as plt


def bmatrix(a):
"""Returns a LaTeX bmatrix
Retrieved from https://stackoverflow.com/questions/17129290/numpy-2d-and-1d-array-to-latex-bmatrix
:a: numpy array
:returns: LaTeX bmatrix as a string
"""
if len(a.shape) > 2:
raise ValueError('bmatrix can at most display two dimensions')
lines = str(a).replace('[', '').replace(']', '').splitlines()
rv = [r'\[']
rv += [r'\begin{bmatrix}']
rv += [' ' + ' & '.join(l.split()) + r'\\' for l in lines]
rv += [r'\end{bmatrix}']
rv += [r'\]\par']
return '\n'.join(rv)


# set a seed
np.random.seed(0)

X = np.array([[0, 0]])

while np.linalg.matrix_rank(X) != X.shape[1]:
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)

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)
n_runs = 11
```

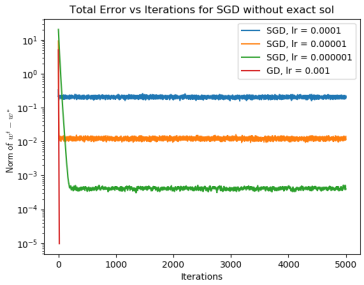
```
w_guesses = [np.zeros((X.shape[1], 1)) for _ in range(n_runs)]
n = X.shape[0]
error = []
it = 0
above_threshold = True
previous_w = np.array(w_guesses)
batch_ize = 1 # as is pointed out on Piazza
while it < max_iterations and above_threshold:
it += 1
curr_error = 0
for j in range(len(w_guesses)):
if gd:
# Your code, implement the gradient for GD
# sample_gradient = ?
sample_gradient = 2 / X.shape[0] * (X.T.dot(X).dot(w_guesses[j]) - X.T.dot(y))
else:
# Your code, implement the gradient for SGD
# sample_gradient = ?
sample_idxes = np.random.choice(X.shape[0], batch_ize, replace=False)
sampleX = X[sample_idxes, :]
sampleY = y[sample_idxes, :]
sample_gradient = 2 / batch_ize * (sampleX.T.dot(sampleX).dot(w_guesses[j]) - sampleX.T.dot(sampleY))

# Your code: implement the gradient update
# learning rate at this step is given by step_size_func(it)
# w_guesses[j] = ?
w_guesses[j] -= step_size_func(it) * sample_gradient

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

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


its = 5000
w_guesses, error = sgd(X, y, w, 1e-10, its, 0.0001)

plt.figure()
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()
plt.savefig('Figure_2f-1.png')
plt.close()

print("Required iterations: " + str(len(error)) + '\par')
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses])
print("Final average error: ", bmatrix(average_error))

y2 = y + np.random.normal(scale=5, size = y.shape)
w=np.linalg.inv(X.T @ X) @ X.T @ y2


its = 5000
w_guesses2, error2 = sgd(X, y2, w, 1e-5, its, 0.0001)
```

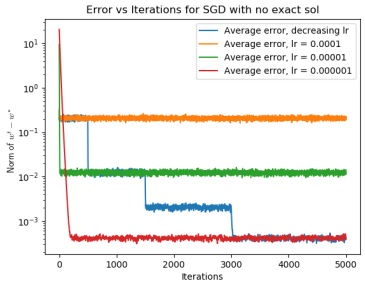
```
w_guesses3, error3 = sgd(X, y2, w, 1e-5, its, 0.00001)
w_guesses4, error4 = sgd(X, y2, w, 1e-5, its, 0.000001)

w_guess_gd, error_gd = sgd(X, y2, w, 1e-5, its, 0.001, True)

plt.figure()
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 = 0.001")
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()
plt.savefig('Figure_2f-2.png')
plt.close()

print("Required iterations, lr = 0.0001: " + str(len(error2)) + '\par')
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses2])
print("Final average error: ", bmatrix(average_error))

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

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

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

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)

plt.figure()
plt.semilogy([i for i in range(len(error_variable))], error_variable, label="Average 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=True)
plt.title("Error vs Iterations for SGD with no exact sol")
plt.legend()
#plt.show()
plt.savefig('Figure_2f-3.png')
plt.close()

print("Required iterations, variable lr: " + str(len(error_variable)) + '\par')
average_error = np.mean([np.linalg.norm(w-w_guess) for w_guess in w_guesses_variable])
```

```
print("Average error with decreasing lr:", bmatrix(average_error))
```

On the other hand, if we do it as mini-batch and keep the same epoch, we will get the following results. The above conclusions still hold true.







Required iterations: 7
Final average error:

$$\begin{bmatrix} 2.63205745954e-13 \end{bmatrix}$$

Required iterations, lr = 0.0001: 5000
Final average error:

$$\begin{bmatrix} 0.220082525024 \end{bmatrix}$$

Required iterations, lr = 0.00001: 5000
Final average error:

$$\begin{bmatrix} 0.014144356575 \end{bmatrix}$$

Required iterations, lr = 0.000001: 5000
Final average error:

$$\begin{bmatrix} 0.000401068466506 \end{bmatrix}$$

Required iterations, GD: 18
Final average error:

$$\begin{bmatrix} 9.71658899539e-06 \end{bmatrix}$$

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

$$\begin{bmatrix} 0.000440791881113 \end{bmatrix}$$

(g) (Bonus) **Prove that there is some constant S and k_0 such that if the learning rate is decaying, the error Δ_k satisfies $\Delta_k \leq \frac{S}{k+k_0}$.** In this part, prove for the case of $M_g = 0$ and $B \neq 0$.

Hint: induction is your friend.

$$\begin{aligned}
&\Delta_k \\
\leq &(1 + \alpha_{k-1}^2 M_g^2 - 2\alpha_{k-1} m)\Delta_{k-1} + \alpha_{k-1}^2 B^2 \\
\leq &(1 - 2\alpha_{k-1} m)\Delta_{k-1} + \alpha_{k-1}^2 B^2
\end{aligned}$$

Let's denote $S = \max\{\Delta_0, \frac{B^2}{m^2}\}$ and $k_0 = 1$. It's easy to see that $\Delta_0 \leq \frac{S}{0+k_0}$. we could choose a step size that is very similar to the one in the lecture $\alpha_{k-1} = \frac{1}{mk}$ and we can simplify the formula above: Let's use mathematical induction and assume the formula below is true for k-1:

$$\Delta_{k-1} \leq \frac{S}{k-1+k_0} = \frac{\max\{\Delta_0, \frac{B^2}{m^2}\}}{k}$$

$$\begin{aligned}
&\Delta_k \\
&\leq (1 - 2\alpha_{k-1} m)\, \Delta_{k-1} + \alpha_{k-1}^2 B^2 \\
&\leq \left(1 - 2m\frac{1}{mk}\right) \Delta_{k-1} + \left(\frac{1}{mk}\right)^2 B^2 \\
&= \left(1 - \frac{2}{k}\right) \Delta_{k-1} + \left(\frac{1}{k}\right)^2 \frac{B^2}{m^2} \\
&\leq \left(1 - \frac{2}{k}\right) \frac{S}{k} + \left(\frac{1}{k}\right)^2 S \\
&= \left(\frac{1}{k} - \frac{1}{k^2}\right) S \\
&= \left(\frac{k-1}{k^2}\right) S \\
&\leq \left(\frac{k-1}{k^2-1}\right) S \\
&= \left(\frac{1}{k+1}\right) S
\end{aligned}$$

Therefore, we have proved that:

$$\Delta_k \leq \frac{S}{k+k_0}$$
$$\text{where} \quad S = \max\{\Delta_0, \frac{B^2}{m^2}\} \quad k_0 = 1$$

Reference :here

(h) (Bonus) **Prove that the same conclusion holds for the case of $M_g \neq 0$ and $B \neq 0$, for some different constant M_g and B.**

Hint: this general case can be reduced to the case above.

The only difference is in the constant. If our step size is small enough we always have $\Delta_k \leq \Delta_0$:

$$\begin{aligned}
&\Delta_k \\
\leq &(1 + \alpha_{k-1}^2 M_g^2 - 2\alpha_{k-1} m)\Delta_{k-1} + \alpha_{k-1}^2 B^2 \\
= &(1 - 2\alpha_{k-1} m)\Delta_{k-1} + \alpha_{k-1}^2 (B^2 + M_g^2 \Delta_{k-1}) \\
\leq &(1 - 2\alpha_{k-1} m)\Delta_{k-1} + \alpha_{k-1}^2 (B^2 + M_g^2 \Delta_0)
\end{aligned}$$

So the above inequality holds that:

$$\Delta_k \leq \frac{S}{k + k_0}$$

$$\text{where} \quad S = \max\{\Delta_0, \frac{B^2 + M_g^2 \Delta_0}{m^2}\} \quad k_0 = 1$$

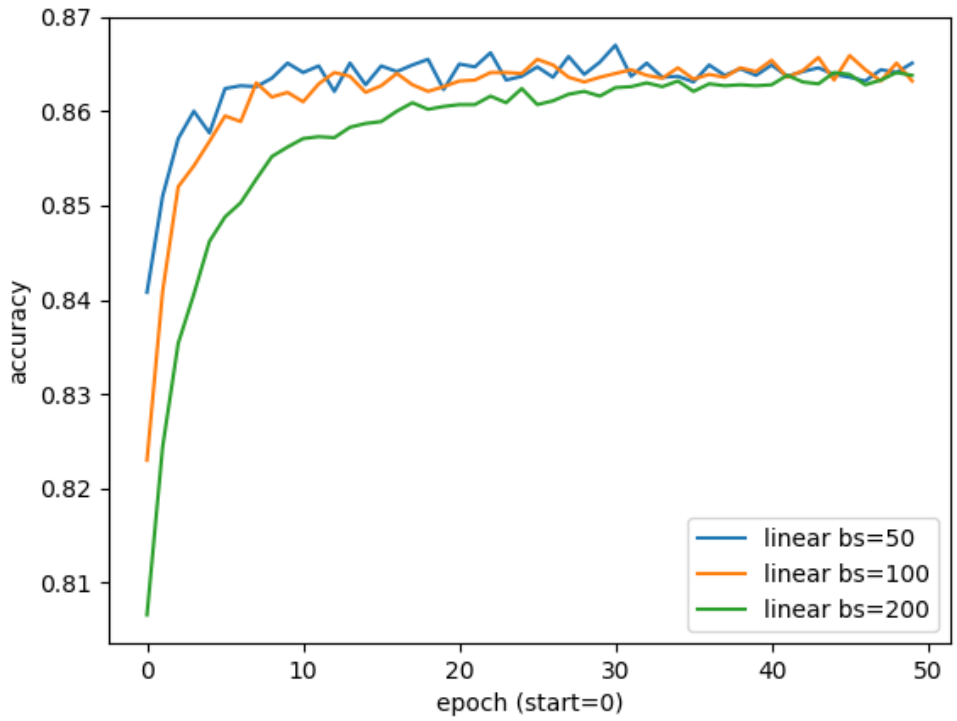
Question 3. Gradient Descent Framework

In HW1, you modeled the classification of digit numbers of the MNIST dataset as a linear regression problem and solved it using its closed-form solution. In this homework, you will model it better by using classification models such as logistic regression and neural networks, and solve it using stochastic gradient descent. The goal of this problem is to show the power of modern machine learning frameworks such as TensorFlow and PyTorch, and how they can solve a wide range of problems. TensorFlow is the recommended framework in this homework and we also provide the starter kit in PyTorch.

(a) The starter code contains an implementation of linear regression for the MNIST dataset to classify 10 digits. Let $\vec{x}_i$ be the feature vector and $\vec{y}_i$ be a one-hot vector encoding of its class, i.e., $y_{i,j} = 1$ if and only if ith images is in class j and $y_{i,j} = 0$ if not. In order to use linear regression to solve a multi-class classification, we will use the *one-vs-all* strategy here. In particular, for each j we will fit a linear regression model $(\vec{w}_j, b_j)$ to minimize $\sum_i (\vec{w}_j^\top \vec{x}_i + b_j - y_{i,j})_2^2$. Then for any image $\vec{x}$, the prediction of its class will be $\arg\max_j (\vec{w}_j^\top \vec{x} + b_j)$.

Read the implementation and run it with batch size equal to 50, 100, and 200. **Attach the "epoch vs validation accuracy" plot. Report the running time for all the batch sizes. Explain the potential benefits and drawbacks of using small batch size, i.e., SGD vs GD.**

Running time for all the batch sizes:
train_linear with a batch size of 50 finishes in 34.232s
train_linear with a batch size of 100 finishes in 21.761s
train_linear with a batch size of 200 finishes in 17.482s



The smaller batch size, the longer it takes to run SGD. This is because we keep epoch the same for different batch sizes. Therefore, the smaller the batch size, the more iterative loops we have to

run in Python. The computation of gradients in a batch, however, is vectorized and the actual loops are run in C/C++ or in parallel which is much faster.

The smaller batch size, the faster it reaches the steady state (finds an optimal solution). However, all batch sizes in this case reach approximately the same accuracy level.

As is pointed out by a GSI on piazza, in practice, a smaller batch size tend to find a better local minimum. However, smaller step size will increase the variance of gradients.

On the other hand, larger step size runs faster but might be less accurate. A more detailed information can be found here.

GD will run similar to SGD with a batch size the same as sample size. Therefore, it runs the slowest and might converge much slower but gives the most accurate prediction.

```
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)


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
# copied from tf.nn.softmax
logits = (tf.matmul(x, W) + b)
pred = tf.exp(logits - tf.reduce_max(logits)) / tf.reduce_sum(tf.exp(logits - tf.reduce_max(logits)))
# copied from https://github.com/tensorflow/tensorflow/issues/2462
# loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduction_indices=[1]))
# modified it to be the stable version
loss = tf.reduce_mean(-tf.log(tf.reduce_sum(y * pred, reduction_indices=[1])))
#################

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=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
logits = tf.matmul(tf.tanh(tf.matmul(x, W1) + b1), W2) + b2
# copied from tf.nn.softmax
pred = tf.exp(logits - tf.reduce_max(logits)) / tf.reduce_sum(tf.exp(logits - tf.reduce_max(logits)))
# copied from https://github.com/tensorflow/tensorflow/issues/2462
# loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduction_indices=[1]))
# modified it to be the stable version
loss = tf.reduce_mean(-tf.log(tf.reduce_sum(y * pred, reduction_indices=[1])))
#################

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


def main():

plt.figure()
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.ylabel('accuracy')
plt.xlabel('epoch (start=0)')
plt.savefig('Figure_3a-linear.png')
plt.close()

plt.figure()
acc_logistic = train_logistic()
plt.plot(acc_logistic, label="logistic regression")
plt.legend()
#plt.show()
plt.ylabel('accuracy')
plt.xlabel('epoch (start=0)')
plt.savefig('Figure_3a-logistic.png')
```

```
    plt.close()

    plt.figure()
    acc_nn = train_nn()
    plt.plot(acc_nn, label="neural network")
    plt.legend()
    plt.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    #plt.show()
    plt.savefig('Figure_3a-neural.png')
    plt.close()

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

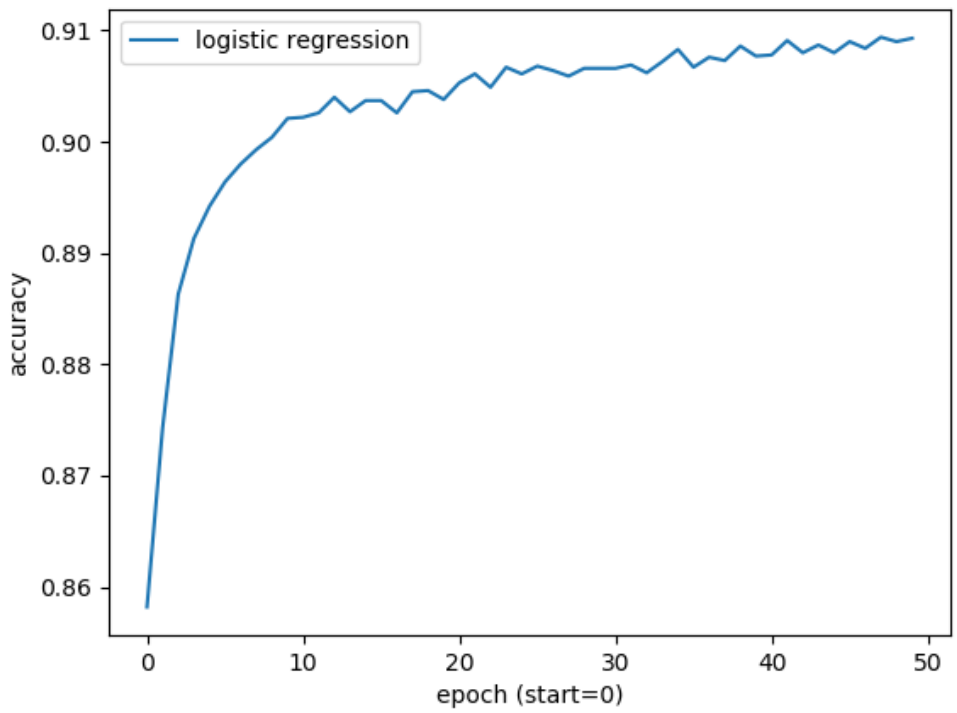
(b) **Implement the `train_logistic` function to do the multi-class logistic regression using softmax function. Attach the plot of the "epoch vs validation accuracy" curve.**. The loss function of the multi-class logistic regression is

$$l = -\sum_{i=1}^{n} \log \left[\textbf{softmax}\left(\mathbf{W}\vec{x}_i + \vec{b}\right)^\top \vec{y}_i \right] \tag{4}$$

where the softmax function $\textbf{softmax} : \mathbb{R}^d \to \mathbb{R}^d$ is defined as

$$\text{softmax}(F)_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)} = \frac{\exp(z_j - z')}{\sum_k \exp(z_k - z')}. \tag{5}$$

Here $z' = \max_j z_j$. The expression on the right is a numerical stable formula to compute the softmax. You may NOT use any functions in `tf.nn.*`, `tf.losses.*`, and `torch.nn.*` for all the parts of this problem.




```
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):
```

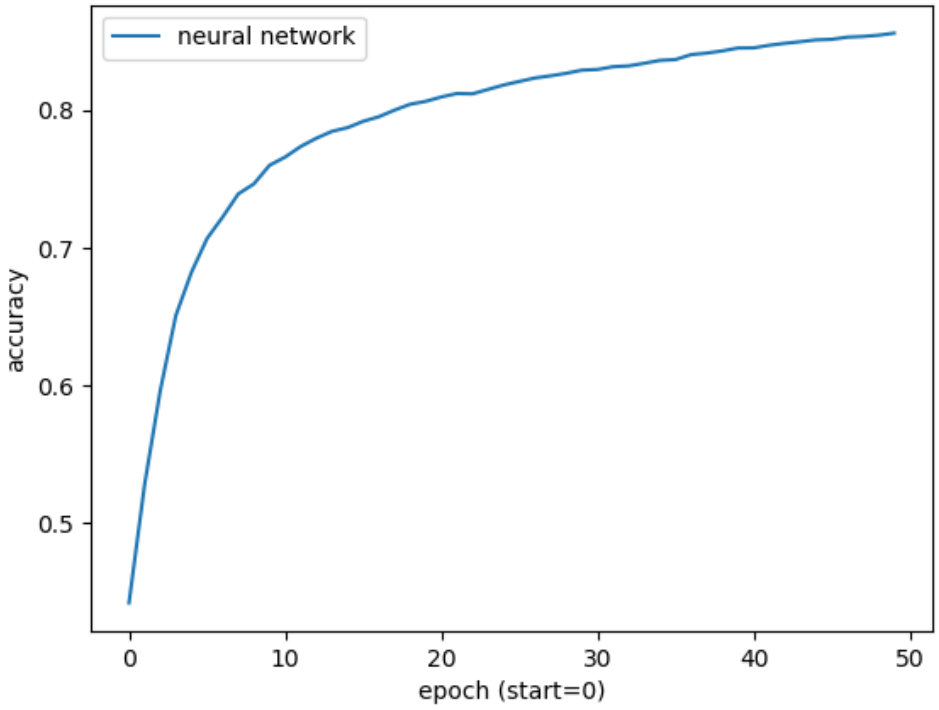
```
acc = []
with tf.Session() as sess: # start training
sess.run(tf.global_variables_initializer()) # Run the initializer
for epoch in range(training_epochs): # Training cycle
avg_loss = 0.
total_batch = int(mnist.train.num_examples / batch_size)
for i in range(total_batch):
batch_xs, batch_ys = mnist.train.next_batch(batch_size)
_, 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)


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
# copied from tf.nn.softmax
logits = (tf.matmul(x, W) + b)
pred = tf.exp(logits - tf.reduce_max(logits)) / tf.reduce_sum(tf.exp(logits - tf.reduce_max(logits)))
# copied from https://github.com/tensorflow/tensorflow/issues/2462
# loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduction_indices=[1]))
# modified it to be the stable version
loss = tf.reduce_mean(-tf.log(tf.reduce_sum(y * pred, reduction_indices=[1])))
################

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=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
    logits = tf.matmul(tf.tanh(tf.matmul(x, W1) + b1), W2) + b2
    # copied from tf.nn.softmax
    pred = tf.exp(logits - tf.reduce_max(logits)) / tf.reduce_sum(tf.exp(logits - tf.reduce_max(logits)))
    # copied from https://github.com/tensorflow/tensorflow/issues/2462
    # loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduction_indices=[1]))
    # modified it to be the stable version
    loss = tf.reduce_mean(-tf.log(tf.reduce_sum(y * pred, reduction_indices=[1])))
    #################

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


    def main():

    plt.figure()
    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.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    plt.savefig('Figure_3a-linear.png')
    plt.close()

    plt.figure()
    acc_logistic = train_logistic()
    plt.plot(acc_logistic, label="logistic regression")
    plt.legend()
    #plt.show()
    plt.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    plt.savefig('Figure_3a-logistic.png')
    plt.close()

    plt.figure()
    acc_nn = train_nn()
    plt.plot(acc_nn, label="neural network")
    plt.legend()
    plt.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    #plt.show()
    plt.savefig('Figure_3a-neural.png')
    plt.close()

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

(c) Copy your code from `train_logistic` to `train_nn` and add an additional tanh nonlinear layer to it. You loss function should be something like

$$l = -\sum_i \log \left[\mathbf{softmax}\left(\mathbf{W}^{(2)} \mathbf{tanh}\left(\mathbf{W}^{(1)} \vec{x}_i \right) + \vec{b} \right)^\top \vec{y}_i \right]. \tag{6}$$

Attach the plot of the "epoch vs validation accuracy" curve. You have the freedom but **are NOT required to** to choose the hyper-parameters to get the best performance.




```
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)


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
    # copied from tf.nn.softmax
    logits = (tf.matmul(x, W) + b)
    pred = tf.exp(logits - tf.reduce_max(logits)) / tf.reduce_sum(tf.exp(logits - tf.reduce_max(logits)))
    # copied from https://github.com/tensorflow/tensorflow/issues/2462
    # loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduction_indices=[1]))
    # modified it to be the stable version
    loss = tf.reduce_mean(-tf.log(tf.reduce_sum(y * pred, reduction_indices=[1])))
    ################

    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=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
    logits = tf.matmul(tf.tanh(tf.matmul(x, W1) + b1), W2) + b2
    # copied from tf.nn.softmax
    pred = tf.exp(logits - tf.reduce_max(logits)) / tf.reduce_sum(tf.exp(logits - tf.reduce_max(logits)))
    # copied from https://github.com/tensorflow/tensorflow/issues/2462
    # loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduction_indices=[1]))
    # modified it to be the stable version
    loss = tf.reduce_mean(-tf.log(tf.reduce_sum(y * pred, reduction_indices=[1])))
```

```
    ################

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


    def main():

    plt.figure()
    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.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    plt.savefig('Figure_3a-linear.png')
    plt.close()

    plt.figure()
    acc_logistic = train_logistic()
    plt.plot(acc_logistic, label="logistic regression")
    plt.legend()
    #plt.show()
    plt.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    plt.savefig('Figure_3a-logistic.png')
    plt.close()

    plt.figure()
    acc_nn = train_nn()
    plt.plot(acc_nn, label="neural network")
    plt.legend()
    plt.ylabel('accuracy')
    plt.xlabel('epoch (start=0)')
    #plt.show()
    plt.savefig('Figure_3a-neural.png')
    plt.close()

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

(d) In our previous lecture, we learned how to solve the total least squares by doing the SVD decomposition. Now we will try to model the problem from the probabilistic perspective and solve it using stochastic gradient descent. Let the probabilistic model be

$$y_i - z_y = \vec{w}^\top (\vec{x}_i - \vec{z}_x) \tag{7}$$

where $\vec{x}_i$ and y_i is the observed data, $y_i - z_y$ is y_{true}, $\vec{x}_i - z_{\vec{x}}$ is x_{true}, $z_y \sim \mathcal{N}(0,1)$, and $z_{\vec{x},j} \sim \mathcal{N}(0,1)$ for all j. **Prove that the log-likelihood for the model in Equation** (7) **is**

$$\sum_{i=1}^{n} \log P_{\vec{w}}(y_i \mid \vec{x}_i) = C - \frac{n}{2} \log\left(\|\vec{w}\|_2^2 + 1\right) - \frac{1}{2(\|\vec{w}\|_2^2+1)} \sum_{i=1}^{n} (y_i - \vec{w}^\top \vec{x}_i)^2, \tag{8}$$

where n is the number of samples, and C is a constant that is not related to $\vec{w}$, y, and $\vec{x}$. *Note that the maximum likelihood estimation of this probabilistic model is not the same as the SVD solution of TLS as we did in lecture.*

$$\begin{aligned} y_i - z_y &= \vec{w}^\top (\vec{x}_i - \vec{z}_x) \\ y_i - \vec{w}^\top \vec{x}_i &= z_y - \vec{w}^\top \vec{z}_x \end{aligned}$$

Now we look at the variance of $z_y - \vec{w}^\top \vec{z}_x$:

$$\begin{aligned} & z_y - \vec{w}^\top \vec{z}_x \\ =& \begin{bmatrix} 1 & -w^\top \end{bmatrix} \begin{bmatrix} z_y \\ \vec{z}_x \end{bmatrix} \\ \sim& \begin{bmatrix} 1 & -w^\top \end{bmatrix} \begin{bmatrix} 1 \\ -w^\top \end{bmatrix} = \|\vec{w}\|_2^2 + 1 \end{aligned}$$

Therefore, we have a new random variable $(z_y - \vec{w}^\top \vec{z}_x) \sim N(0, (\|\vec{w}\|_2^2 + 1))$

$$\begin{aligned} & \log P_{\vec{w}}(y_i \mid \vec{x}_i) \\ =& \log\left(\frac{1}{\sqrt{2\pi(\|\vec{w}\|_2^2+1)}} e^{-\frac{(y_i - \vec{w}^\top \vec{x}_i)^2}{2(\|\vec{w}\|_2^2+1)}} \right) \\ =& \log(\frac{1}{\sqrt{2\pi}}) - \frac{1}{2} \log\left(\|\vec{w}\|_2^2 + 1\right) - \frac{1}{2(\|\vec{w}\|_2^2+1)} (y_i - \vec{w}^\top \vec{x}_i)^2 \end{aligned}$$

We sum up over n data points to get the following equation:

$$\sum_{i=1}^{n} \log P_{\vec{w}}(y_i \mid \vec{x}_i) = C - \frac{n}{2} \log\left(\|\vec{w}\|_2^2 + 1\right) - \frac{1}{2(\|\vec{w}\|_2^2+1)} \sum_{i=1}^{n} (y_i - \vec{w}^\top \vec{x}_i)^2,$$

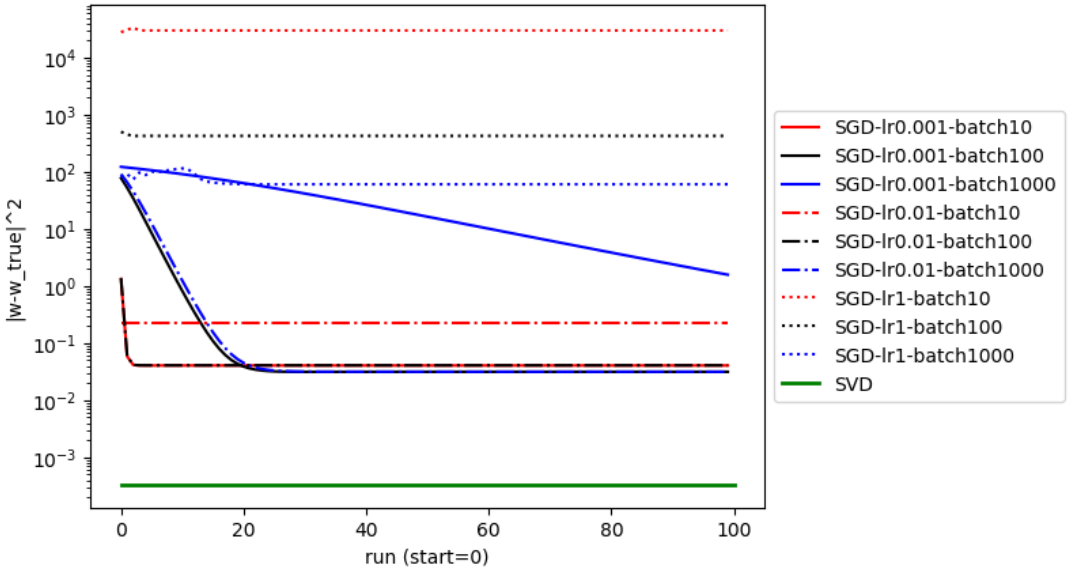
(e) **Implement the stochastic gradient descent to find the MLE for the model above.** In the starter code, we generate some data for you and you will need to recover $\vec{w}$ using the observed data. **Report the error** $\|w^* - w_{\text{true}}\|_2^2$ where w^* is the one that you recover. Try to play with hyper-parameters such as the batch size and the learning rate. **Is the solution of SGD sensitive to its hyper-parameters in this problem?**

First of all, SGD cannot find the same solution as SVD and the overall average error is higher for SGD.

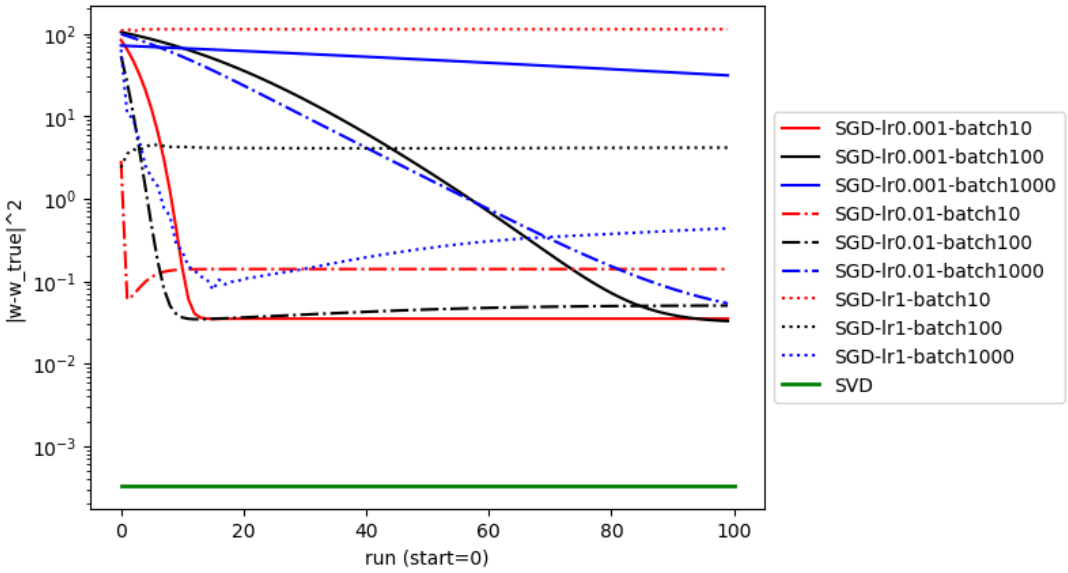
Second, SGD is sensitive to learning rate but if the learning rate is small enough, SGD is not sensitive to batch size. To be more specific, a smaller batch size actually performs better.On the other hand, as is shown by the dotted lines, if learning rate is too high, the solution doesn't converge to the same one and it is affected by the learning rate.

If ADO is used, things get a little worse sometimes. The reason behind it is not clear to me because I don't understand how AdamOptimizer works.

When GradientDescentOptimizer is used:



When the fancy AdamOptimizer is used:



Here are the errors I got by running the script using AdamOptimizer

Epoch: 0001 cost= 176.475031026 $\|w - w_{true}\|^2 = 65.601266212$
Epoch: 0002 cost= 103.951681646 $\|w - w_{true}\|^2 = 34.170639800$
Epoch: 0003 cost= 51.933814780 $\|w - w_{true}\|^2 = 14.881957584$
Epoch: 0004 cost= 23.555888176 $\|w - w_{true}\|^2 = 5.730799898$
Epoch: 0005 cost= 11.350280762 $\|w - w_{true}\|^2 = 2.105033012$
Epoch: 0006 cost= 6.853173089 $\|w - w_{true}\|^2 = 0.783763169$
Epoch: 0007 cost= 5.303149267 $\|w - w_{true}\|^2 = 0.308513649$
Epoch: 0008 cost= 4.778555353 $\|w - w_{true}\|^2 = 0.135571334$
Epoch: 0009 cost= 4.604046047 $\|w - w_{true}\|^2 = 0.071969237$
Epoch: 0010 cost= 4.548503510 $\|w - w_{true}\|^2 = 0.048356726$
Epoch: 0011 cost= 4.532444425 $\|w - w_{true}\|^2 = 0.039451412$
Epoch: 0012 cost= 4.528880044 $\|w - w_{true}\|^2 = 0.036029853$
Epoch: 0013 cost= 4.529018656 $\|w - w_{true}\|^2 = 0.034724194$
Epoch: 0014 cost= 4.530174359 $\|w - w_{true}\|^2 = 0.034278651$
Epoch: 0015 cost= 4.531587517 $\|w - w_{true}\|^2 = 0.034201705$
Epoch: 0016 cost= 4.533060205 $\|w - w_{true}\|^2 = 0.034288427$
Epoch: 0017 cost= 4.534545171 $\|w - w_{true}\|^2 = 0.034449246$
Epoch: 0018 cost= 4.536031572 $\|w - w_{true}\|^2 = 0.034644697$
Epoch: 0019 cost= 4.537517444 $\|w - w_{true}\|^2 = 0.034857193$
Epoch: 0020 cost= 4.539001604 $\|w - w_{true}\|^2 = 0.035079454$
Epoch: 0021 cost= 4.540484730 $\|w - w_{true}\|^2 = 0.035308480$
Epoch: 0022 cost= 4.541966403 $\|w - w_{true}\|^2 = 0.035543212$
Epoch: 0023 cost= 4.543446632 $\|w - w_{true}\|^2 = 0.035782859$
Epoch: 0024 cost= 4.544925078 $\|w - w_{true}\|^2 = 0.036027385$
Epoch: 0025 cost= 4.546401143 $\|w - w_{true}\|^2 = 0.036276613$
Epoch: 0026 cost= 4.547874431 $\|w - w_{true}\|^2 = 0.036530100$
Epoch: 0027 cost= 4.549344301 $\|w - w_{true}\|^2 = 0.036787879$
Epoch: 0028 cost= 4.550809868 $\|w - w_{true}\|^2 = 0.037049667$
Epoch: 0029 cost= 4.552270476 $\|w - w_{true}\|^2 = 0.037315199$

Epoch: 0030 cost= 4.553724869 $\|w - w_{true}\|^2 = 0.037584206$
Epoch: 0031 cost= 4.555172225 $\|w - w_{true}\|^2 = 0.037856620$
Epoch: 0032 cost= 4.556611653 $\|w - w_{true}\|^2 = 0.038131914$
Epoch: 0033 cost= 4.558041875 $\|w - w_{true}\|^2 = 0.038409819$
Epoch: 0034 cost= 4.559461772 $\|w - w_{true}\|^2 = 0.038690132$
Epoch: 0035 cost= 4.560870107 $\|w - w_{true}\|^2 = 0.038972552$
Epoch: 0036 cost= 4.562265992 $\|w - w_{true}\|^2 = 0.039256616$
Epoch: 0037 cost= 4.563647985 $\|w - w_{true}\|^2 = 0.039542303$
Epoch: 0038 cost= 4.565014970 $\|w - w_{true}\|^2 = 0.039828978$
Epoch: 0039 cost= 4.566365794 $\|w - w_{true}\|^2 = 0.040116413$
Epoch: 0040 cost= 4.567699540 $\|w - w_{true}\|^2 = 0.040404214$
Epoch: 0041 cost= 4.569014974 $\|w - w_{true}\|^2 = 0.040692077$
Epoch: 0042 cost= 4.570310787 $\|w - w_{true}\|^2 = 0.040979781$
Epoch: 0043 cost= 4.571586100 $\|w - w_{true}\|^2 = 0.041266884$
Epoch: 0044 cost= 4.572840385 $\|w - w_{true}\|^2 = 0.041552981$
Epoch: 0045 cost= 4.574071900 $\|w - w_{true}\|^2 = 0.041837775$
Epoch: 0046 cost= 4.575280428 $\|w - w_{true}\|^2 = 0.042121047$
Epoch: 0047 cost= 4.576464470 $\|w - w_{true}\|^2 = 0.042402396$
Epoch: 0048 cost= 4.577623928 $\|w - w_{true}\|^2 = 0.042681493$
Epoch: 0049 cost= 4.578757974 $\|w - w_{true}\|^2 = 0.042958011$
Epoch: 0050 cost= 4.579865591 $\|w - w_{true}\|^2 = 0.043231635$
Epoch: 0051 cost= 4.580946636 $\|w - w_{true}\|^2 = 0.043502063$
Epoch: 0052 cost= 4.582000383 $\|w - w_{true}\|^2 = 0.043769053$
Epoch: 0053 cost= 4.583026516 $\|w - w_{true}\|^2 = 0.044032195$
Epoch: 0054 cost= 4.584024553 $\|w - w_{true}\|^2 = 0.044291321$
Epoch: 0055 cost= 4.584994626 $\|w - w_{true}\|^2 = 0.044546112$
Epoch: 0056 cost= 4.585936034 $\|w - w_{true}\|^2 = 0.044796378$
Epoch: 0057 cost= 4.586849050 $\|w - w_{true}\|^2 = 0.045041798$
Epoch: 0058 cost= 4.587733352 $\|w - w_{true}\|^2 = 0.045282368$
Epoch: 0059 cost= 4.588589136 $\|w - w_{true}\|^2 = 0.045517536$
Epoch: 0060 cost= 4.589416488 $\|w - w_{true}\|^2 = 0.045747282$
Epoch: 0061 cost= 4.590215266 $\|w - w_{true}\|^2 = 0.045971696$
Epoch: 0062 cost= 4.590986152 $\|w - w_{true}\|^2 = 0.046190227$
Epoch: 0063 cost= 4.591729073 $\|w - w_{true}\|^2 = 0.046402930$
Epoch: 0064 cost= 4.592444432 $\|w - w_{true}\|^2 = 0.046609749$
Epoch: 0065 cost= 4.593132591 $\|w - w_{true}\|^2 = 0.046810466$
Epoch: 0066 cost= 4.593794191 $\|w - w_{true}\|^2 = 0.047005180$
Epoch: 0067 cost= 4.594429219 $\|w - w_{true}\|^2 = 0.047193719$
Epoch: 0068 cost= 4.595038644 $\|w - w_{true}\|^2 = 0.047376018$
Epoch: 0069 cost= 4.595622845 $\|w - w_{true}\|^2 = 0.047552205$
Epoch: 0070 cost= 4.596182152 $\|w - w_{true}\|^2 = 0.047722237$
Epoch: 0071 cost= 4.596717564 $\|w - w_{true}\|^2 = 0.047886134$
Epoch: 0072 cost= 4.597229449 $\|w - w_{true}\|^2 = 0.048043941$
Epoch: 0073 cost= 4.597718751 $\|w - w_{true}\|^2 = 0.048195661$
Epoch: 0074 cost= 4.598185778 $\|w - w_{true}\|^2 = 0.048341436$
Epoch: 0075 cost= 4.598631386 $\|w - w_{true}\|^2 = 0.048481327$
Epoch: 0076 cost= 4.599056172 $\|w - w_{true}\|^2 = 0.048615482$

Epoch: 0077 cost= 4.599460562 $\|w - w_{true}\|^2 = 0.048744025$
Epoch: 0078 cost= 4.599846033 $\|w - w_{true}\|^2 = 0.048867088$
Epoch: 0079 cost= 4.600212614 $\|w - w_{true}\|^2 = 0.048984539$
Epoch: 0080 cost= 4.600561170 $\|w - w_{true}\|^2 = 0.049096832$
Epoch: 0081 cost= 4.600892373 $\|w - w_{true}\|^2 = 0.049204037$
Epoch: 0082 cost= 4.601207030 $\|w - w_{true}\|^2 = 0.049306219$
Epoch: 0083 cost= 4.601505705 $\|w - w_{true}\|^2 = 0.049403621$
Epoch: 0084 cost= 4.601789153 $\|w - w_{true}\|^2 = 0.049496384$
Epoch: 0085 cost= 4.602057652 $\|w - w_{true}\|^2 = 0.049584527$
Epoch: 0086 cost= 4.602312398 $\|w - w_{true}\|^2 = 0.049668484$
Epoch: 0087 cost= 4.602553570 $\|w - w_{true}\|^2 = 0.049748116$
Epoch: 0088 cost= 4.602782011 $\|w - w_{true}\|^2 = 0.049823787$
Epoch: 0089 cost= 4.602998237 $\|w - w_{true}\|^2 = 0.049895569$
Epoch: 0090 cost= 4.603202883 $\|w - w_{true}\|^2 = 0.049963744$
Epoch: 0091 cost= 4.603396380 $\|w - w_{true}\|^2 = 0.050028255$
Epoch: 0092 cost= 4.603579076 $\|w - w_{true}\|^2 = 0.050089449$
Epoch: 0093 cost= 4.603752176 $\|w - w_{true}\|^2 = 0.050147334$
Epoch: 0094 cost= 4.603915564 $\|w - w_{true}\|^2 = 0.050202154$
Epoch: 0095 cost= 4.604069781 $\|w - w_{true}\|^2 = 0.050254091$
Epoch: 0096 cost= 4.604215411 $\|w - w_{true}\|^2 = 0.050303011$
Epoch: 0097 cost= 4.604352824 $\|w - w_{true}\|^2 = 0.050349316$
Epoch: 0098 cost= 4.604482476 $\|w - w_{true}\|^2 = 0.050393141$
Epoch: 0099 cost= 4.604604884 $\|w - w_{true}\|^2 = 0.050434444$
Epoch: 0100 cost= 4.604720350 $\|w - w_{true}\|^2 = 0.050473477$
TLS through SVD error: $\|w - w_{true}\|^2 = 0.0003320206394634698$

Code to report the error:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

n_data = 6000
n_dim = 50

np.random.seed(0)

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 = tf.log(tf.norm(w) + 1) / 2 + \
1 / (2 * (tf.norm(w) + 1)) * tf.reduce_mean((tf.matmul(x, w) - y) ** 2)
################

errors_sgd = np.zeros(training_epochs)

# 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)
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()
errors_sgd[epoch] = np.sum((w_sgd - w_true) ** 2)
print("Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(avg_cost),
"$\|w-w_{{true}}\|^2 = {:.9f}".format(np.sum((w_sgd - w_true) ** 2)) + '$ \par')

#print("Epoch:", '%04d' % (training_epochs), "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))

plt.figure()
plt.semilogy(errors_sgd, label="SGD")
plt.semilogy([0, len(errors_sgd)], [errors_sgd[-1], errors_sgd[-1]], label="last error (SGD)",
    linewidth=2)
plt.text(len(errors_sgd) * 0.6, errors_sgd[-1] * 2, str(errors_sgd[-1]), fontsize=12)
plt.semilogy([0, len(errors_sgd)], [error, error], label="SVD", linewidth=2)
plt.text(len(errors_sgd) * 0.6, error * 2, str(error), fontsize=12)
plt.legend()
plt.ylabel('|w-w_true|^2')
plt.xlabel('run (start=0)')
# plt.show()
plt.savefig('Figure_3e-lr'+str(learning_rate) + '-batch' + str(batch_size) + '.png')
plt.close()


if __name__ == "__main__":
```

```
tf.set_random_seed(0)
np.random.seed(0)
main()
```

Code to vary hyperparameters:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

n_data = 6000
n_dim = 50

np.random.seed(0)

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 tabu(sizes, widths, error):
print(r'\begin{tabu} to 1.0\textwidth { ' + ''.join('|X[c] ' * (len(sizes)+1)) + ' | }')
print(r'\hline')
header = '{:^8}'
for _ in range(len(sizes)):
header += ' {:^8}'
header += ' {:^8}'
headerText = [' '] + ['& ' + str(s) for s in sizes] + [' \\\\']
print(header.format(*headerText))
for width, row in zip(widths, error):
text = '{:>8}'
for _ in range(len(row)):
text += ' {:<8}'
text += ' {:<8}'
#'{0:.1f}'.format(r)
rowText = [str(width)] + ['& ' + str(r) for r in row] + [' \\\\']
print(r'\hline')
print(text.format(*rowText))
print(r'\hline')
print(r'\end{tabu}\par')
print(r'\hfill \par')


def main():
lrs = [0.001, 0.01, 1]
batches = [10, 100, 1000]
lrs_lines = ['-', '-.', ':']
batches_colors = ['r', 'k', 'b']
plt.figure()
last_erros_sgd = np.zeros((len(lrs), len(batches)))

for ilr, learning_rate in enumerate(lrs):
for ibatch, batch_size in enumerate(batches):
```

```
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 = tf.log(tf.norm(w) + 1) / 2 + \
1 / (2 * (tf.norm(w) + 1)) * tf.reduce_mean((tf.matmul(x, w) - y) ** 2)
################
errors_sgd = np.zeros(training_epochs)

# 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)
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()
errors_sgd[epoch] = np.sum((w_sgd - w_true) ** 2)
plt.semilogy(errors_sgd, batches_colors[ibatch]+lrs_lines[ilr], label='SGD-lr' + str(learning_rate) +
    '-batch' + str(batch_size))
last_erros_sgd[ilr, ibatch] = errors_sgd[-1]
#plt.semilogy([0, len(errors_sgd)], [errors_sgd[-1], errors_sgd[-1]], label="last error (SGD)",
#           linewidth=2)
#plt.text(len(errors_sgd) * 0.6, errors_sgd[-1] * 2, str(errors_sgd[-1]), fontsize=12)
# print("Epoch:", '%04d' % (epoch + 1), "cost=", "{:.9f}".format(avg_cost),
#     "|w-w_true|^2 = {:.9f}".format(np.sum((w_sgd - w_true) ** 2)))

# print("Epoch:", '%04d' % (training_epochs), "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))

plt.semilogy([0, len(errors_sgd)], [error, error], 'g', label="SVD", linewidth=2)
# plt.text(len(errors_sgd) * 0.6, error * 2, str(error), fontsize=12)
plt.legend(loc='upper center', bbox_to_anchor=(0.5, 1.05),
ncol=3, fancybox=True, shadow=False)
plt.ylabel('|w-w_true|^2')
plt.xlabel('run (start=0)')
# plt.show()
```

```
    plt.savefig('Figure_3e-2.png')
    plt.xlim([0, training_epochs * 1.5])
    plt.close()

    print('SGD errors with different learning rates and batch sizes:')
    tabu(lrs, batches, last_erros_sgd)


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

Question 4. [BONUS] Genome-Wide Association Study

All the following text is present in the accompanying jupyter notebook, but the notebook has additional explanations and accompanying figures and code. We recommend you do not read this pdf, but go directly to the jupyter notebook. This pdf was included for quick reference.

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. **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 $\mathbf{X} \in \mathbb{R}^{n \times m}$, 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 i^{th} person at genetic feature j. [1] 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 modelling, we would erroneously conclude that the data were full of signal.*

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.

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, $\texttt{diff\_ll} = LL_j - LL_{-j}$–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).

[1] Technically, the entries of the matrix correspond to having zero, one or two mutant versions of the DNA, but we will treat them as real-valued in this problem.

Figure 1: Example of quantile-quantile plot that shows large deviation from expectation.

(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 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 $Unif[0, 1]$.

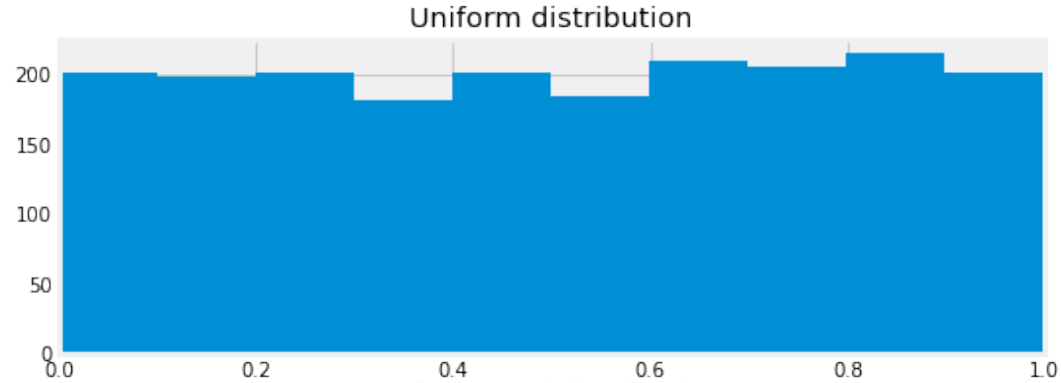
Use the qqplot function to make a qq-plot for each of the 3 distributions provided below and explain your findings. What should we observe in our qq-plot if our empirical distribution looks more and more similar to a Unif[0, 1]?. Note that we use two kinds of qq-plots: one in p-value space and one negative 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.

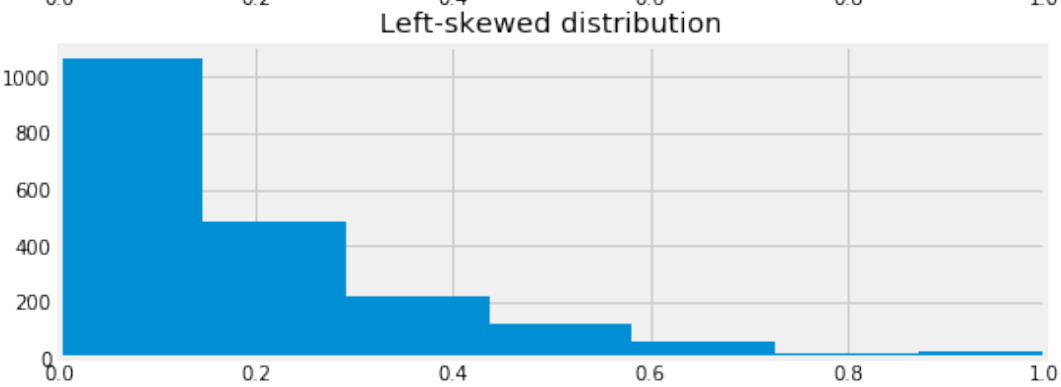
(Note: the left-skewed and right-skewed graphs are messed up so the terms are interpreted in the opposite way in the answer)

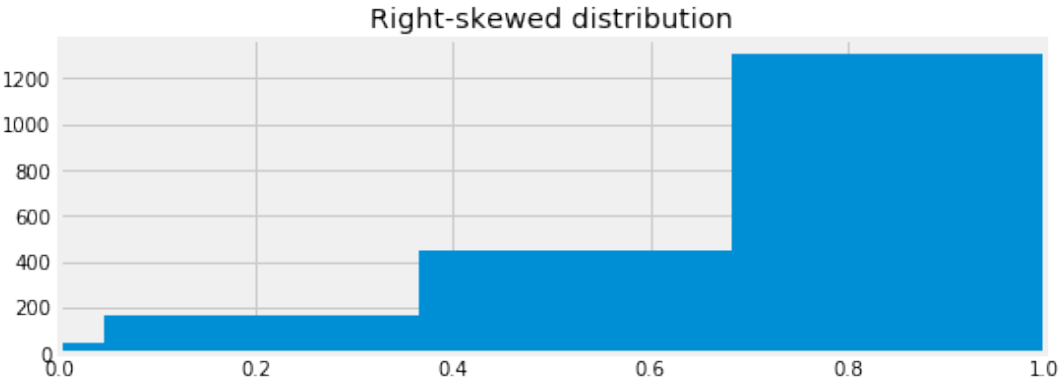
If our p-value is uniformly distributed $p \sim \text{Unif}[0, 1]$, we will see all points lie on the diagonal line, which means our observation is the same as our null hypothesis.

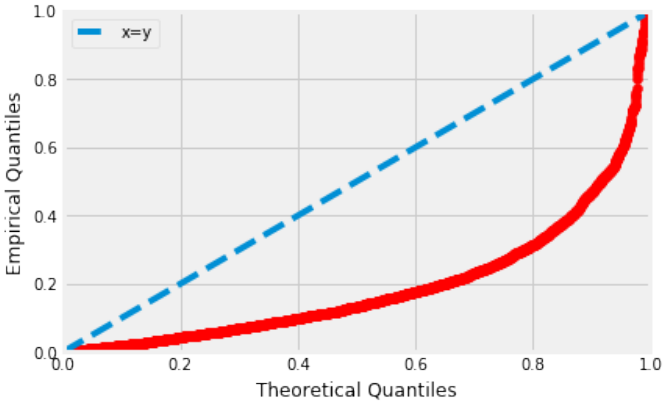
If our p-value is very small for many features (left-skewed), we will see many points lie below the diagonal line on the linear-scaled plot and above the diagonal line on the negative log-scaled plot, which means many genetic features are associated with the trait.

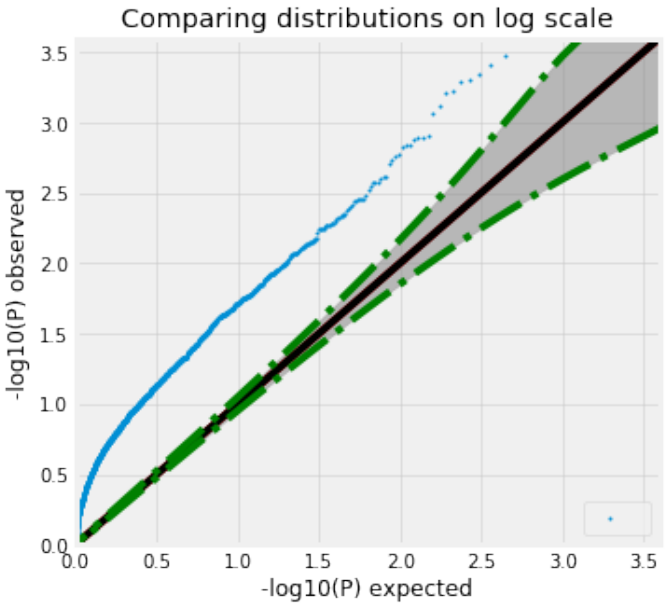
If our p-value is very small for a small subset of features (right-skewed), we will see many points lie above the diagonal line on the linear-scaled plot and below the diagonal line on the negative log-scaled plot, which means a few genetic features are associated with the trait and they could be critical.

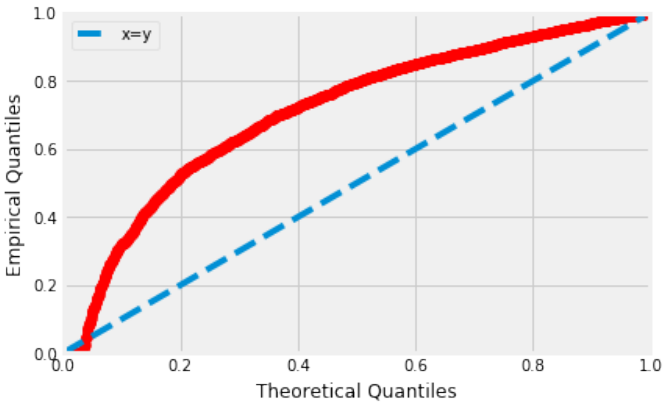

Uniform distribution
200
150
100
50
0
0.0
0.2
0.4
0.6
0.8
1.0

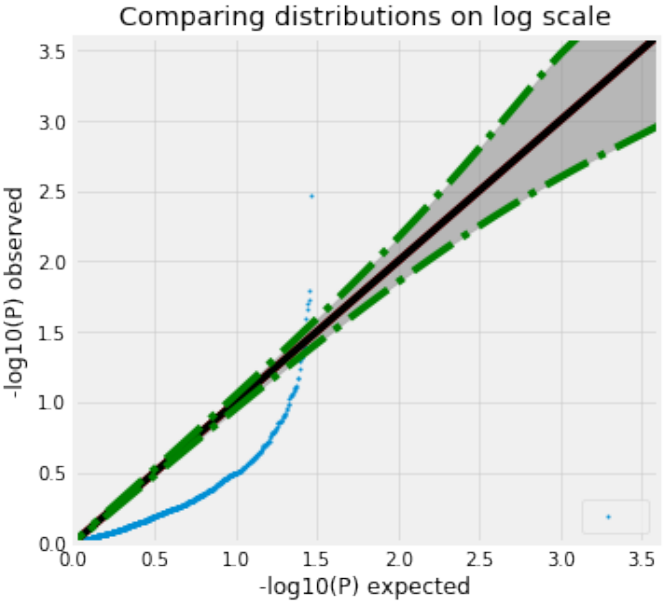

Left-skewed distribution
1000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0

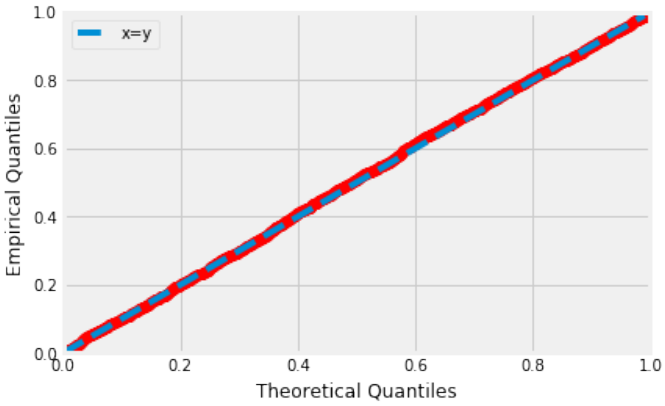

Right-skewed distribution
1200
1000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0

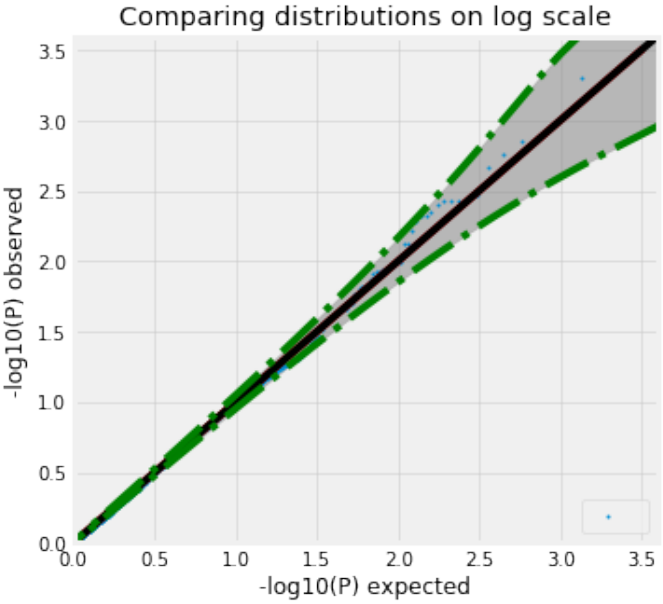

x=y
Empirical Quantiles
Theoretical Quantiles
0.0
0.2
0.4
0.6
0.8
1.0


Comparing distributions on log scale
-log10(P) observed
-log10(P) expected
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


x=y
Empirical Quantiles
Theoretical Quantiles
0.0
0.2
0.4
0.6
0.8
1.0


Comparing distributions on log scale
-log10(P) observed
-log10(P) expected
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

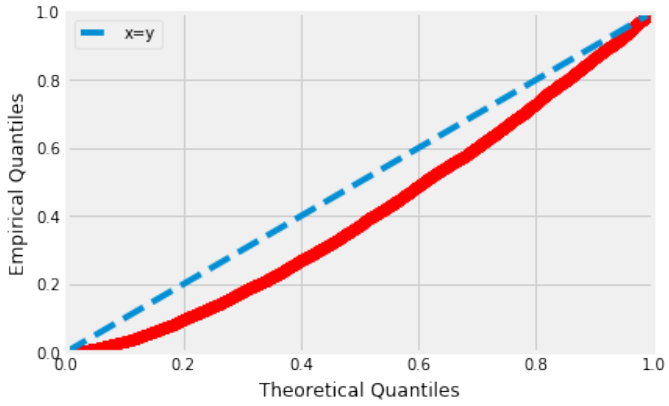

x=y
Empirical Quantiles
Theoretical Quantiles
0.0
0.2
0.4
0.6
0.8
1.0

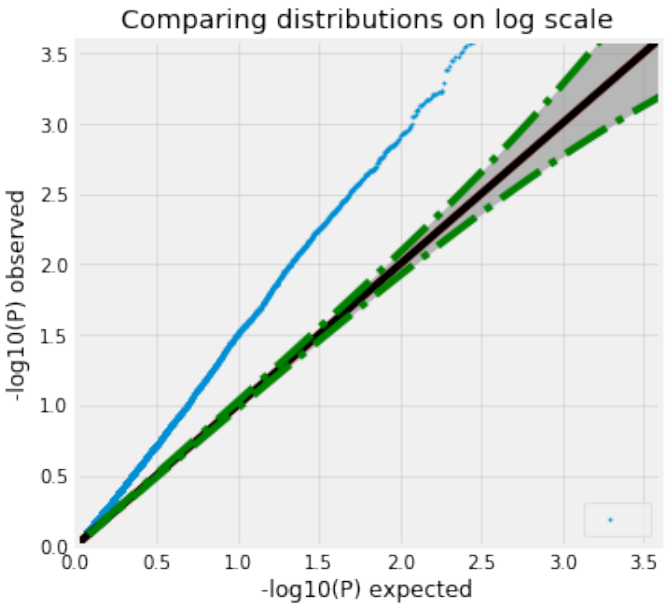

Comparing distributions on log scale
-log10(P) observed
-log10(P) expected
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

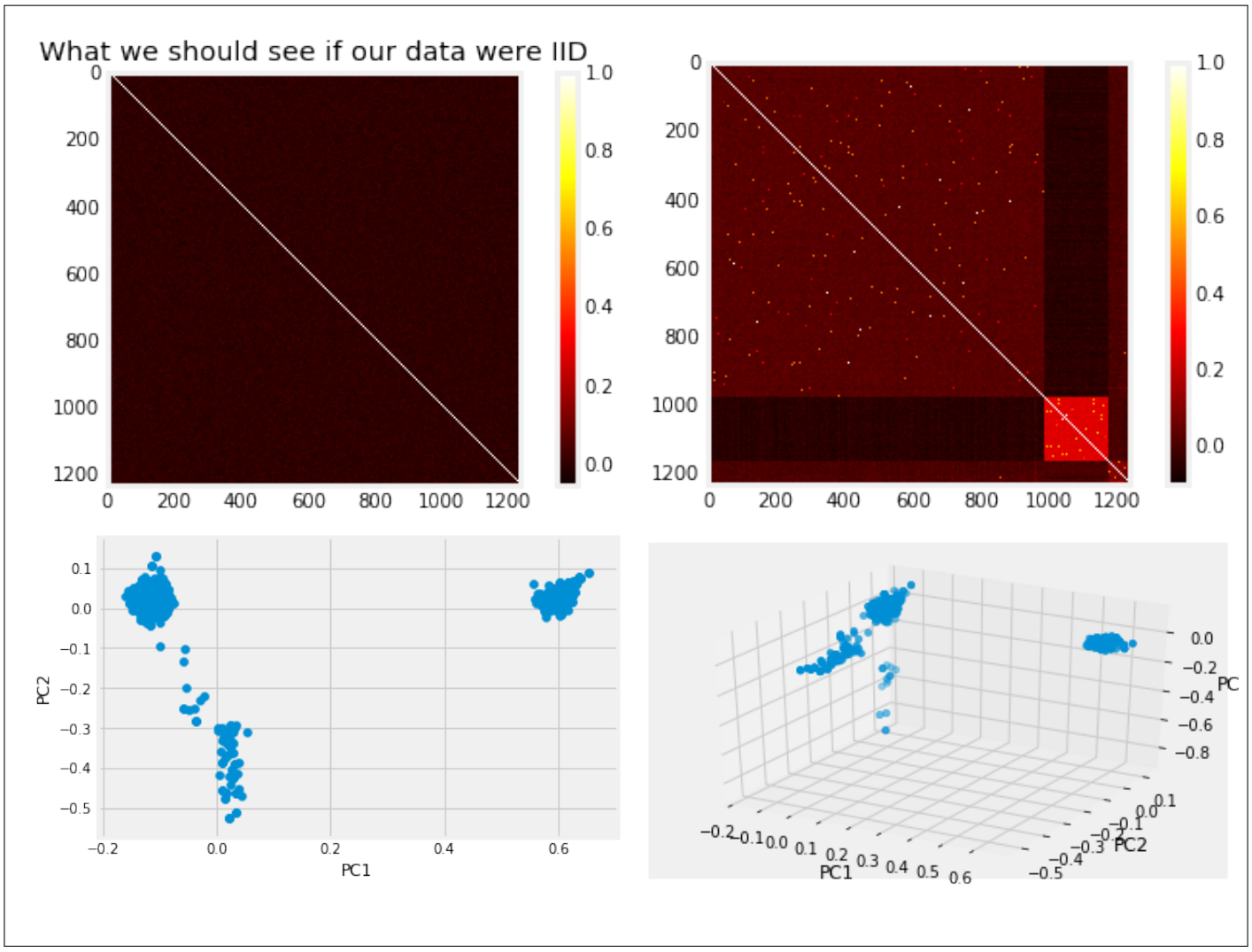
(b) We will use linear models in the genetic marker we are testing. In particular, when testing the j^{th} genetic feature, we have that the trait, $\vec{y}$, is a linear function of the genetic variant, i.e. $\vec{y} = \vec{x_j} w_1 + \vec{w_0} + \vec{\epsilon}$ where ϵ_i is random noise distributed as $N(0, \sigma^2)$, $w_1 \in \mathbb{R}^1$, $\vec{w_0} \in \mathbb{R}^{n \times 1}$ is a constant vector, and $\vec{x_j}$ is the jth column of $\mathbf{X}$, representing data for the jth genetic variant. To simplify matters, we will add a column of ones to the right end of $\vec{x_j}$ and rewrite the regression as $\vec{y} = [\vec{x_j}, \vec{1}]\vec{w} + \vec{\epsilon}$ where $[\vec{x_j}, \vec{1}]$ is the jth column of $\mathbf{X}$ 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 $\vec{y} = [\vec{x_j}, \vec{1}]\vec{w} + \vec{\epsilon}$ where $\vec{x_j}$ 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 describe, 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?**

We can see most features are associated with the trait because qqplot shows the same pattern as the left-skewed distribution above. This indicates that there could be some external relationships among the individuals in each group other than the trait (smoking) we are interested in. Therefore, the association we see might not be the association we are looking for but due to some other factors. These factors are not removed or considered because our linear regression model assume individual data points are iid.

Therefore, we can conclude that the naive linear regression approach is not appropriate.






What we should see if our data were IID
PC1
PC2
PC

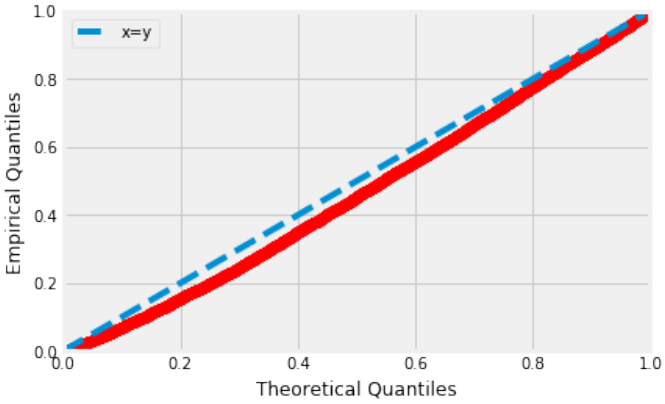
(c) From the quantile-quantile plot in the previous part, it appears that the model is picking up on more association than theoretically expected. The reason for this is owing to the assumption of iid noise being correct. In particular, this data set contains individuals from different racial backgrounds, 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.

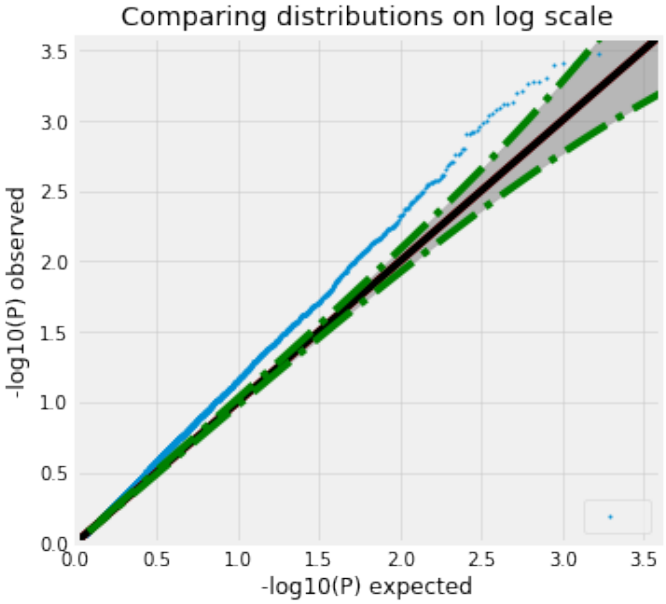
So instead of using all genetic features, we will try using PCA to reduce the number of genetic features. As we saw in class, PCA on a genetic similarity matrix can capture geography, which correlates to race, quite well. So instead of adding all the genetic features, we will instead use only three features[2], $\mathbf{X_{proj}}$, which are the $\mathbf{X}$ projected onto the top 3 principal components of $\mathbf{X}$. Consequently, the updated null model is $\vec{y} = \mathbf{X_{proj}}\vec{w_{proj}} + \vec{\epsilon}$ where $\vec{w_{proj}} \in \mathbb{R}^{3\times 1}$, while the alternative model is $\vec{y} = [\vec{x_j}, \mathbf{X_{proj}}, \vec{1}]\vec{w} + \vec{\epsilon}$ 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?**

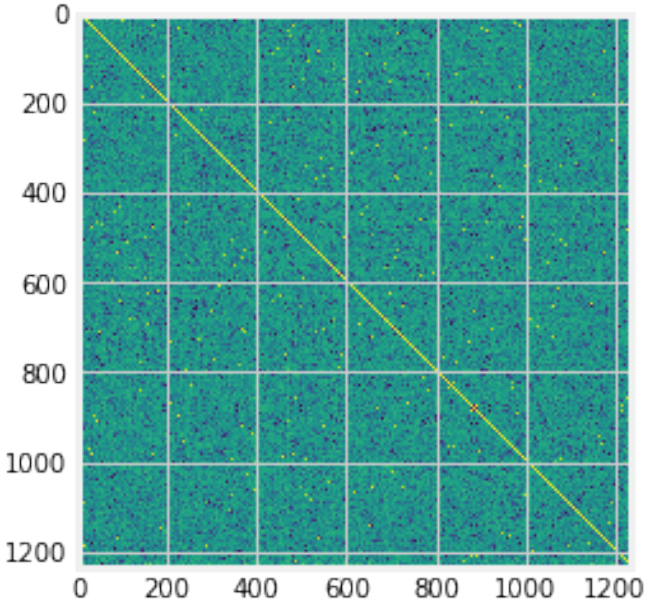
We don't want to use all genetic features so we can avoid over-fitting. Because There are roughly 1300 individuals and 7500 genetic variants, if we used all genetic variants then our linear model is guaranteed to over-fit because the number of features is larger than the number of data points.

The qqplot of PCA linear regression approach shows a curve that is much closer to the null hypothesis. However, there are still many significant features. This tells us that using the first principle components can filter out some relationships we are not interested in but not all. This might be due to the removal of other principle components that might contribute to other relationships among individuals.

[2]One needs to choose this number, but we have done so for you.


x=y
Empirical Quantiles
Theoretical Quantiles
0.0
0.2
0.4
0.6
0.8
1.0


Comparing distributions on log scale
-log10(P) observed
-log10(P) expected
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


0
200
400
600
800
1000
1200

(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 $\vec{y} \sim N(y|[\vec{x_j}, 1]\vec{w}, I\sigma^2 + \mathbf{X}\mathbf{X}^\top\sigma_k^2)$.

Specifically, $\vec{y} = [\vec{x_j}, 1]\vec{w} + \vec{z} + \vec{\epsilon}$ where $\vec{z} \sim N(0, \sigma_k^2\mathbf{K})$ where $\mathbf{K} = \mathbf{X}\mathbf{X}^\top$, σ_k, $\vec{w} \in \mathbb{R}^{m\times 1}$, and σ are parameters we want to estimate. Notice that $\vec{y} \sim N([\vec{x_j}, 1]\vec{w}, \sigma^2 I + \sigma_k^2 K)$. 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 $\mathbf{K} = \mathbf{U}\mathbf{D}\mathbf{U}^\top$, 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.

We can use a similar approach to kernalized-PCA as what we derived in the midterm with a small tweak. To make the multivariate Gaussian of $\vec{z}$ diagonal, we need to apply a linear transformation to $\vec{z}$. Recall that:

$$\begin{aligned} \mathbf{A}\vec{z} &\sim N(0, \mathbf{A}\Sigma\mathbf{A}^\top) \\ \text{where} \quad \vec{z} &\sim N(0, \Sigma) \end{aligned}$$

We want:

$$\mathbf{A}(\sigma^2\mathbf{I} + \sigma_k^2\mathbf{U}\mathbf{D}\mathbf{U}^\top)\mathbf{A}^\top = \sigma^2\mathbf{I} + \sigma_k^2\mathbf{D} \qquad \Rightarrow \qquad \mathbf{A} = \mathbf{U}^\top$$

One way to save computation time is to apply $\mathbf{A}$ first so the covariance matrix is diagonalized and the matrix inversion is just the reciprocal of all diagonal entries.

The other way to look at it is the same as what we did in the midterm, where we have:

$$\begin{aligned} &(\sigma^2\mathbf{I} + \sigma_k^2\mathbf{K})^{-1} \\ =&(\sigma^2\mathbf{I} + \sigma_k^2\mathbf{U}\mathbf{D}\mathbf{U}^\top)^{-1} \\ =&(\mathbf{U}(\sigma^2\mathbf{I} + \sigma_k^2\mathbf{D})\mathbf{U}^\top)^{-1} \\ =&\mathbf{U}(\sigma^2\mathbf{I} + \sigma_k^2\mathbf{D})^{-1}\mathbf{U}^\top \\ =&\sum_{i=1}^{d} \frac{1}{\sigma^2 + \sigma_k^2\sigma_i}\vec{u}_i\vec{u}_i^\top \end{aligned}$$

The determinant can be computer by multiplying all the eigenvalues together:

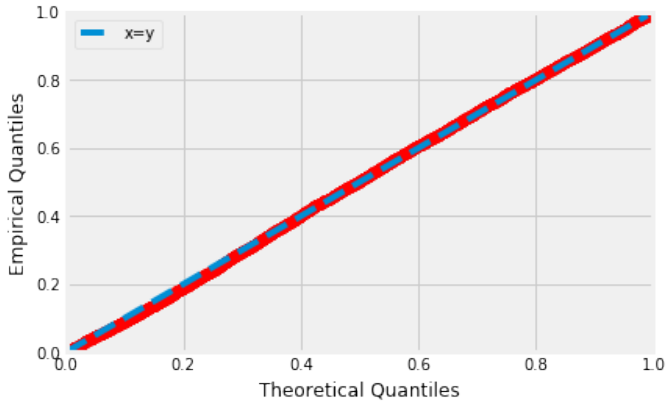
$$\left|\sigma^2\mathbf{I} + \sigma_k^2\mathbf{K}\right| = \prod_{i=1}^{n}\left(\sigma^2 + \sigma_k^2\sigma_i\right)$$

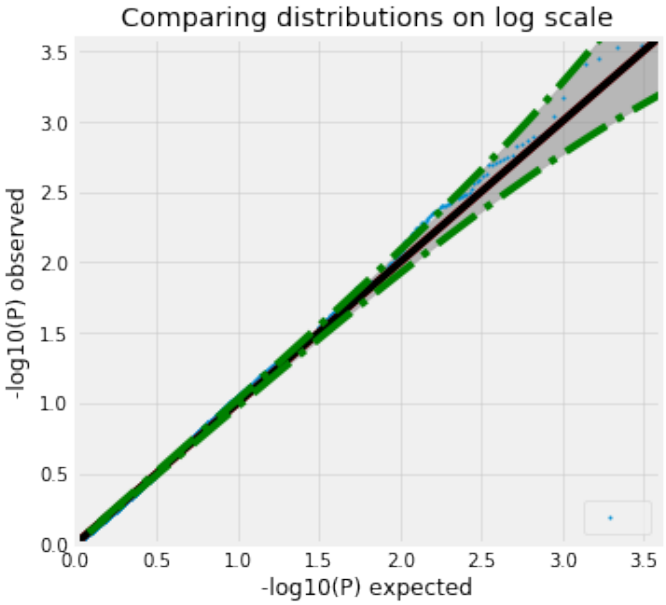
Because we have diagonalized the covariance matrix so we only need $\mathbb{O}(n^2)$ to compute $U^\top \vec{y}$ and $\mathbb{O}(n)$ to computer the inverse or the determinant of the diagonalized covariance matrix.

It would not be computationally more efficient if we only have one genetic feature to test. Because in that case, the number of data points is much larger than the number of features. Using the regular version of linear regression is preferable when $n >> d$.

As we can see from the qqplots, all p-values lie on the diagonal line, which means none of the features is associated with the trait and the empirical p-values are drawn from the uniform distribution Unif$[0, 1]$.

Therefore, we can conclude that there is no genetic marker in this group of individuals that determines whether the individual smoke or not.





Question 5. Your Own Question

Write your own question, and provide a thorough solution.

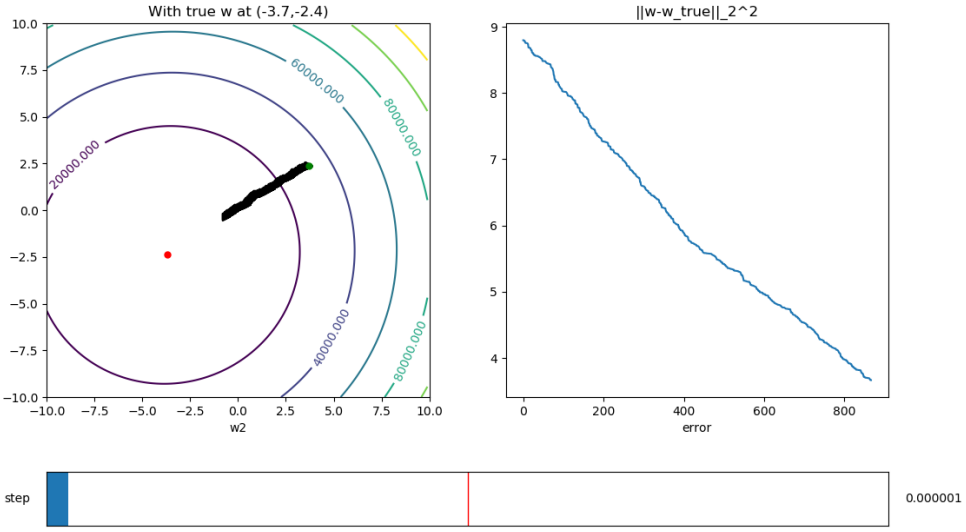
Writing your own problems is a very important way to really learn the material. The famous "Bloom's Taxonomy" that lists the levels of learning is: Remember, Understand, Apply, Analyze, Evaluate, and Create. Using what you know to create is the top-level. We rarely ask you any HW questions about the lowest level of straight-up remembering, expecting you to be able to do that yourself. (e.g. make yourself flashcards) But we don't want the same to be true about the highest level.

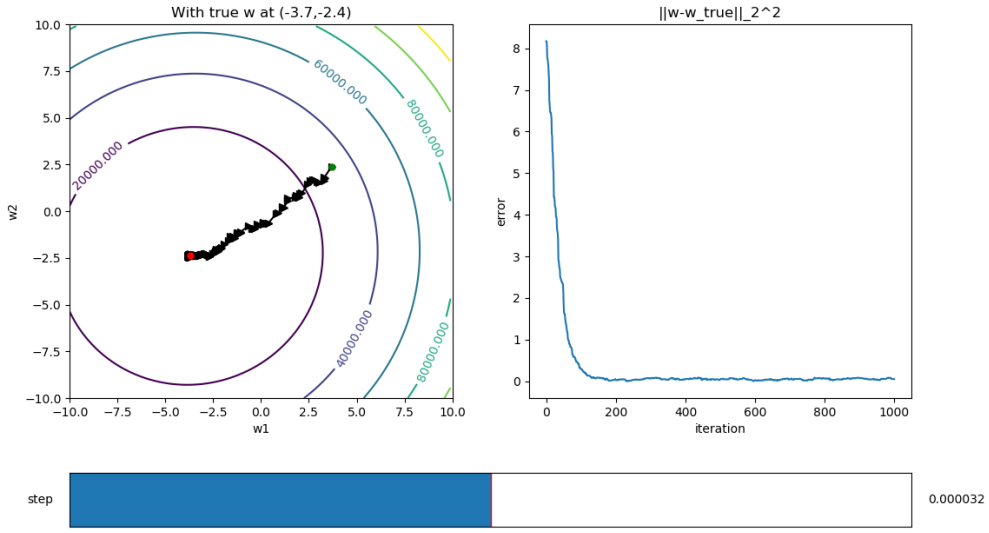
As a practical matter, having some practice at trying to create problems helps you study for exams much better than simply counting on solving existing practice problems. This is because thinking about how to create an interesting problem forces you to really look at the material from the perspective of those who are going to create the exams.

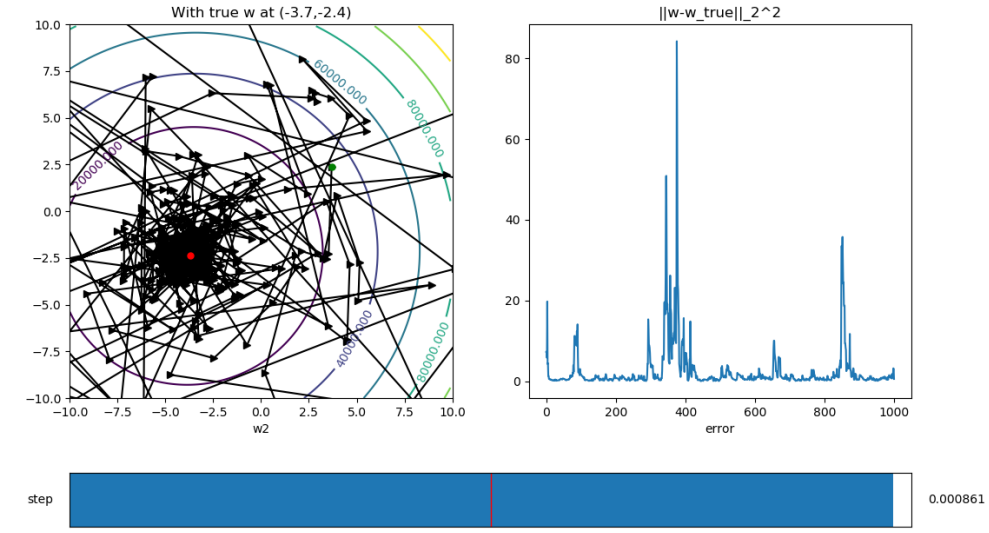
Besides, this is fun. If you want to make a boring problem, go ahead. That is your prerogative. But it is more fun to really engage with the material, discover something interesting, and then come up with a problem that walks others down a journey that lets them share your discovery. You don't have to achieve this every week. But unless you try every week, it probably won't happen ever.

Let's make a dynamic plot to show what will happen if we change our learning rate of SGD and visualize it in Python.

As we can see below, a small learning rate will not reach the optimal point and a big learning rate will not converge. This is the intuitive conclusion we have made.








```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import matplotlib.patches as mpatches
from matplotlib.widgets import Slider


def loss_func(X, w, y):
return np.mean((X.dot(w) - y) ** 2)


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)
n_runs = 1
# w_guesses = [np.zeros((X.shape[1], 1)) for _ in range(n_runs)]
# w_guesses = np.random.normal(scale = 10, size = (n_runs, 2, 1))
w_guesses = - np.tile(w_true.reshape((1,) + w_true.shape), (n_runs, 1, 1))
w_hist = np.tile(w_guesses[0, :, :], (max_iterations + 1, 1, 1))
n = X.shape[0]
error = []
it = 0
above_threshold = True
previous_w = np.array(w_guesses)
batch_size = 1 # as is pointed out on Piazza
while it < max_iterations and above_threshold:
it += 1
curr_error = 0
for j in range(len(w_guesses)):
if gd:
sample_gradient = 2 / n * (X.T.dot(X).dot(w_guesses[j]) - X.T.dot(y))
else:
sample_idxes = np.random.choice(X.shape[0], batch_size, replace=False)
sampleX = X[sample_idxes, :]
sampleY = y[sample_idxes, :]
sample_gradient = 2 / batch_size * (sampleX.T.dot(sampleX).dot(w_guesses[j]) - sampleX.T.dot(sampleY))
w_guesses[j] -= step_size_func(it) * sample_gradient
curr_error += np.linalg.norm(w_guesses[j]-w_actual)
error.append(curr_error / n_runs)
w_hist[it, :, :] = np.mean(np.asarray(w_guesses), axis=0)
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)
w_hist = w_hist[0: it, :, :]
return w_hist, error


# set a seed
np.random.seed(0)

# X = np.array([[0, 0]])
X = np.random.normal(scale=20, size=(100, 2))

# Theoretical optimal solution
w_true = np.random.normal(scale=10, size=(2, 1))
# w_true = np.array([[-5.0, -5.0]]).T
y = X.dot(w_true)
y2 = y + np.random.normal(scale=5, size=y.shape)

limits = [-10.0, 10.0]
x1_step = np.arange(-10.0, 10.0, 0.1)
x2_step = np.arange(-10.0, 10.0, 0.1)
X1, X2 = np.meshgrid(x1_step, x2_step)
Z = np.array([[loss_func(X, np.array([[X1[i, j], X2[i, j]]]).T, y) for j in range(len(x1_step))] \
for i in range(len(x2_step))])

plt.figure(figsize=(12.5, 7.5))
ax1 = plt.subplot2grid((8, 2), (0, 0), rowspan=6)
ax2 = plt.subplot2grid((8, 2), (0, 1), rowspan=6)
ax3 = plt.subplot2grid((8, 2), (7, 0), colspan=2)

step_slider = Slider(ax3, 'step', valmin=-6, valmax=-3,
valinit=-4.5, valfmt="%2f")
step_slider.valtext.set_text('%3f' % np.power(10.0, step_slider.val))
```

```
its = 1000


def slider_update(val):
amp = np.power(10, val)
step_slider.valtext.set_text('%3f' % amp)
plt.sca(ax1)
plt.cla()
CS = plt.contour(X1, X2, Z)
plt.clabel(CS, inline=1, fontsize=10)
plt.title('With true w at (%.1f,%.1f)' % (w_true[0, 0], w_true[1, 0]))
plt.xlabel('w1')
plt.xlabel('w2')
plt.xlim(limits)
plt.ylim(limits)

w_hist, error = sgd(X, y2, w_true, 1e-5, its, amp)

plt.plot(w_hist[:, 0, 0], w_hist[:, 1, 0], 'k-')
plt.plot(w_hist[:, 0, 0], w_hist[:, 1, 0], 'k>')
plt.plot(w_true[0, 0], w_true[1, 0], 'r.', markersize=10)
plt.plot(w_hist[0, 0, 0], w_hist[0, 1, 0], 'g.', markersize=10)

plt.sca(ax2)
plt.cla()
plt.plot(error)
plt.xlabel('iteration')
plt.xlabel('error')
plt.title('||w-w_true||_2^2')


# print(plt.axes())
CS = ax1.contour(X1, X2, Z)

ax1.clabel(CS, inline=1, fontsize=10)
ax1.set_title('With true w at (%.1f,%.1f)' % (w_true[0, 0], w_true[1, 0]))
ax1.set_xlabel('w1')
ax1.set_ylabel('w2')
ax1.set_xlim(limits)
ax1.set_ylim(limits)

w_hist, error = sgd(X, y2, w_true, 1e-5, its, np.power(10.0, step_slider.val))

ax1.plot(w_hist[:, 0, 0], w_hist[:, 1, 0], 'k-')
ax1.plot(w_hist[:, 0, 0], w_hist[:, 1, 0], 'k>')
ax1.plot(w_true[0, 0], w_true[1, 0], 'r.', markersize=10)
ax1.plot(w_hist[0, 0, 0], w_hist[0, 1, 0], 'g.', markersize=10)

ax2.plot(error)
ax2.set_xlabel('iteration')
ax2.set_ylabel('error')
ax2.set_title('||w-w_true||_2^2')


# step_slider.valtext.set_visible(True)
step_slider.on_changed(slider_update)

plt.show()
```