Foundations of Data Science and Machine Learning – *Homework 5*

Isaac Martin

Last compiled April 14, 2023

EXERCISE 1. Suppose $\mathbf{A}$ is a $n \times d$ full-rank matrix, with $n < d$, and fix $\mathbf{b} \in \mathbb{R}^n$. Consider minimizing the least squares objective $F(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|_2^2$ Note that in this setting, the solution space $\mathcal{S} = \{x : \mathbf{Ax} = \mathbf{b}\}$ is an affine subspace of $\mathbb{R}^d$. We use gradient descent with constant step-size:

$$\mathbf{x} = \mathbf{x}_{k-1} - \eta \nabla F(\mathbf{x}_{k-1}).$$

(a) Give an upper bound for the step-size η such that gradient descent is guaranteed to converge for η below this threshold.

(b) Suppose that gradient descent is initialized at $\mathbf{x}_0 = 0$. Show that when gradient descent converges, it must converge to the least-norm solution $\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in \mathcal{S}} \|\mathbf{x}\|_2^2$.

Proof:

(a) In class, we showed that if ∇F is Lipschitz with Lipschitz constant L, then choosing $\eta = 1/L$ guarantees the convergence of gradient descent. In particular, any $\eta \leq 1/L$ will guarantee the convergence of gradient descent, so we need only find L. We have

$$\begin{aligned}\|\nabla F(\mathbf{x}) - \nabla F(\mathbf{y})\|_2 &= \|2\mathbf{A}^\top(\mathbf{Ax} - \mathbf{b}) - 2\mathbf{A}^\top(\mathbf{Ay} - \mathbf{b})\|_2 \\ &= \|2\mathbf{A}^\top\mathbf{A}(\mathbf{x} - \mathbf{y})\|_2 \\ &\leq 2\|\mathbf{A}^\top\mathbf{A}\| \cdot \|\mathbf{x} - \mathbf{y}\|_2\end{aligned}$$

where $\|\cdot\|$ denotes the operator norm. Hence choosing $\eta \leq (2\|\mathbf{A}^\top\mathbf{A}\|)^{-1}$ will guarantee the convergence of gradient descent for any initialization.

(b) Let us first prove the hint, namely, that if $\mathbf{x} \in \operatorname{img} \mathbf{A}^\top$ (i.e. if $\mathbf{x}$ is in the rowspan of $\mathbf{A}$) then so is $A\mathbf{x} - \eta\nabla F(\mathbf{x})$. Suppose then that $\mathbf{x} = \mathbf{A}^\top\mathbf{u}$ for some $\mathbf{u} \in \mathbb{R}^n$. Then

$$\begin{aligned}\mathbf{y} = \mathbf{A} \cdot \mathbf{A}^\top\mathbf{u} - \nabla F(\mathbf{A}^\top\mathbf{u}) &= \mathbf{A} \cdot \mathbf{A}^\top\mathbf{u} - \nabla 2A^\top(\mathbf{AA}^\top\mathbf{u} - \mathbf{u}) \\ &= \mathbf{A}^\top u - 2\eta\mathbf{A}^\top\mathbf{AA}^\top\mathbf{u} - 2\mathbf{A}^\top\mathbf{b} \\ &= \mathbf{A}^\top\left(\mathbf{u} - 2\eta\mathbf{AA}^\top\mathbf{u} - 2\mathbf{b}\right) \implies \mathbf{y} \in \operatorname{img}\mathbf{A}^\top.\end{aligned}$$

Because the update rule is continuous and the image of affine linear transformations is closed, we can further conclude that an initialization $\mathbf{x}_0$ is in the rowspan of $\mathbf{A}$ if and only if the point $\mathbf{x}^*$ it converges to is in the rowspan of $\mathbf{A}$, provided η is chosen small enough to guarantee convergence.

Now we prove that any two points initialized in the rowspan of $\mathbf{A}$ converge to the same point. Take $\mathbf{x}_0 = \mathbf{A}^\top\mathbf{u}_0$ to be an initialization for some $\mathbf{u}_0 \in \mathbb{R}^n$. By what we have previously shown, $\mathbf{x}^* = \mathbf{A}^\top\mathbf{u}^*$ for some $\mathbf{u}^* \in \mathbb{R}^n$, supposing we have chosen η to be small enough. Since $\mathbf{x}^*$ is a stable point of the update rule, we get that $\nabla F(\mathbf{x}^*) = 0$ and hence

$$\begin{aligned} \nabla F(\mathbf{A}^\top \mathbf{u}^*) = 2\mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top \mathbf{u}^* - \mathbf{b}) = 0 \\ \implies \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top \mathbf{u}^* - \mathbf{b}) = 0 \\ \implies \mathbf{A}^\top \mathbf{u}^* - \mathbf{b} = 0 \end{aligned}$$

since $\mathbf{A}$ is full rank with $n < d$ (so $\ker A^\top = 0$). This means $\mathbf{u}^* = (\mathbf{A}\mathbf{A}^\top)^{-1}\mathbf{b}$, noting that the inverse $(\mathbf{A}\mathbf{A}^\top)^{-1}$ exists again because $\mathbf{A}$ is fully rank with $n < d$. Using this expression for $\mathbf{u}^*$ gives us that $\mathbf{x}^* = \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)^{-1}\mathbf{b}$, which notably does not depend on the initialization, implying that any two points initialized in the rowspan of $\mathbf{A}$ converge to the same point.

Finally, consider two different initialization $\mathbf{y}_0 \in \mathbb{R}^d \setminus \operatorname{img}(A^\top)$ and $\mathbf{x}_0 \in \operatorname{img}(A^\top)$. As before, $\mathbf{x}_0 = \mathbf{A}^\top \mathbf{u}$ for some $\mathbf{u} \in \mathbb{R}^n$. Since $\mathbf{y}_0$ is not in the rowspan of $\mathbf{A}$, the stable point $\mathbf{y}^*$ of the update rule to which $\mathbf{y}_0$ converges is also not in $\operatorname{img}(\mathbf{A}^\top)$. Hence $\mathbf{y}^* = \mathbf{A}^\top \mathbf{u}^* + \mathbf{v}$ for some $\mathbf{v} \notin \operatorname{img} \mathbf{A}^\top$, where $\mathbf{u}^*$ is as above. The stability condition $\nabla F(\mathbf{y}^*) = 0$ gives us $\mathbf{A}\mathbf{A}^\top \mathbf{u}^* + \mathbf{A}\mathbf{v} - \mathbf{b} = 0$ repeating the calculation from the last paragraph. But $\mathbf{A}\mathbf{A}^\top \mathbf{u}^* - \mathbf{b} = 0$, so $\mathbf{A}\mathbf{v} = 0$. This means

$$\begin{aligned} \|\mathbf{y}^*\|_2^2 &= (\mathbf{A}^\top \mathbf{u}^* + \mathbf{v})^\top (\mathbf{A}^\top \mathbf{u}^* + \mathbf{v}) \\ &= \mathbf{u}^\top \mathbf{A}^\top \mathbf{A}\mathbf{u} + \mathbf{u}^\top \mathbf{A}v + (\mathbf{A}\mathbf{v})^\top \mathbf{u} + \mathbf{v}^\top \mathbf{v} \\ &= \mathbf{u}^\top \mathbf{A}^\top \mathbf{A}\mathbf{u} + 0 + 0 + \mathbf{v}^\top \mathbf{v} \\ &\geq \mathbf{u}^\top \mathbf{A}^\top \mathbf{A}\mathbf{u} \\ &= \|\mathbf{x}^*\|_2^2. \end{aligned}$$

Thus, $\mathbf{x}^* = \arg\min_{\mathbf{x} \in \mathcal{S}} \|\mathbf{x}\|_2^2$. Any point in the rowspan of $\mathbf{A}$ converges to $\mathbf{x}^*$ under the update rule; in particular, the initialization $\mathbf{x}_0 = 0$ converges to $\mathbf{x}^*$, proving the desired result.

□

Exercise 2. Let $f : \mathbb{R}^d \longrightarrow \mathbb{R}$ be a differentiable function. It satisfies the PL-inequality if there exists a constant $\mu > 0$ such that for all $w \in \mathbb{R}^d$ it holds

$$\frac{1}{2}\|\nabla f(w)\|_2^2 \geq \mu(f(w) - f^*).$$

By contrast we say f is *invex* if there exists a function $\eta : \mathbb{R}^d \times \mathbb{R}^d \longrightarrow \mathbb{R}^d$ such that for all $x, y \in \mathbb{R}^d$ it holds

$$f(y) \geq f(x) + \nabla f(x)^\top \eta(x, y).$$

(a) Show that if f satisfies the PL-inequality then f is invex.

(b) Show that any stationary point of an invex function is a global minimizer.

Proof:

(a) Since f satisfies the PL-inequality, for some $\mu > 0$

$$\frac{1}{2}\|\nabla f(w)\|_2^2 \geq \mu(f(w) - f^*)$$

for all $w \in \mathbb{R}^d$, where f^* is the global minimum of f. Rearranging, we get

$$\begin{aligned}
&\frac{1}{2}\|\nabla f(w)\|_2^2 \geq \mu(f(w) - f^*) \\
\implies & \nabla f(w)^\top \nabla f(w) \geq 2\mu f(w) - 2\mu f^* \\
\implies & -\nabla f(w)^\top \nabla f(w) \leq 2\mu f^* - 2\mu f(w) \leq f(w) \leq 2\mu f(u) - 2\mu f(w)
\end{aligned}$$

for any $w, u \in \mathbb{R}^d$, since f^* is the global minimum of f. This in turn implies that

$$2\mu f(w) - \nabla f(w)^\top \nabla f(w) \leq 2\mu f(u),$$

so if we set $\eta(x, y) = -\frac{1}{2\mu}\nabla f(x)$ then we get

$$f(x) = \nabla f(x)^\top \cdot \eta(x, y) \leq f(y)$$

for all $x, y \in \mathbb{R}^d$. This proves that f is invex.

(b) A point $\mathbf{x}$ is a stationary point of f if $\nabla f(x) = 0$. If f is invex, then we get

$$f(y) \geq f(x) + \nabla f(x)^t op \eta(x, y) = f(x)$$

for all $y \in \mathbb{R}^d$. Hence any stationary point of f is a global minima. Combining with part (a) we see that any function which satisfies the PL-inequality is easily optimized.

$\square$

Exercise 3. In your favorite programming language, implement stochastic gradient-descent for the linear least squares loss $f(\mathbf{w}) = \frac{1}{2}\|\mathbf{AW} - \mathbf{b}\|_2^2$. Provide convergence plots to validate the convergence guarantees for SGD discussed in class. Specifically, compare empirical and theoretical convergence rates when $\mathbf{A} \in \mathbb{R}^{10{,}000 \times 1{,}000}$ has iid $\mathcal{N}(0, 1/\sqrt{1000})$ Gaussian entries and $\mathbf{b} = \mathbf{A1} + \varepsilon$ where $\mathbf{1}$ is the all-ones vector and ε has iid Gaussian antries with variance 1, then 0.1, then 0.01 and finally 0. Repeat the comparisons but now consider $\mathbf{A} \in \mathbb{R}^{10000 \times 1000}$ whose jth row has iid $\mathcal{N}(0, 1/\sqrt{1000j})$ Gaussian entries. (*Note your answer should include 8-plots, because there are two choices of* $\mathbf{A}$ *and four different choices of* ε.)

Proof: Here is the first round of graphs with A chosen to be a $10^4 \times 10^3$ matrix with entries normally sampled from $\mathcal{N}(0, 1/\sqrt{10000})$, denoted by A_1 below. The batchsize was chosen to be 100, and sgd was allowed to run for 100 iterations with a step size of 0.1. The theoretical bound is shown in red and the empirical result in blue.

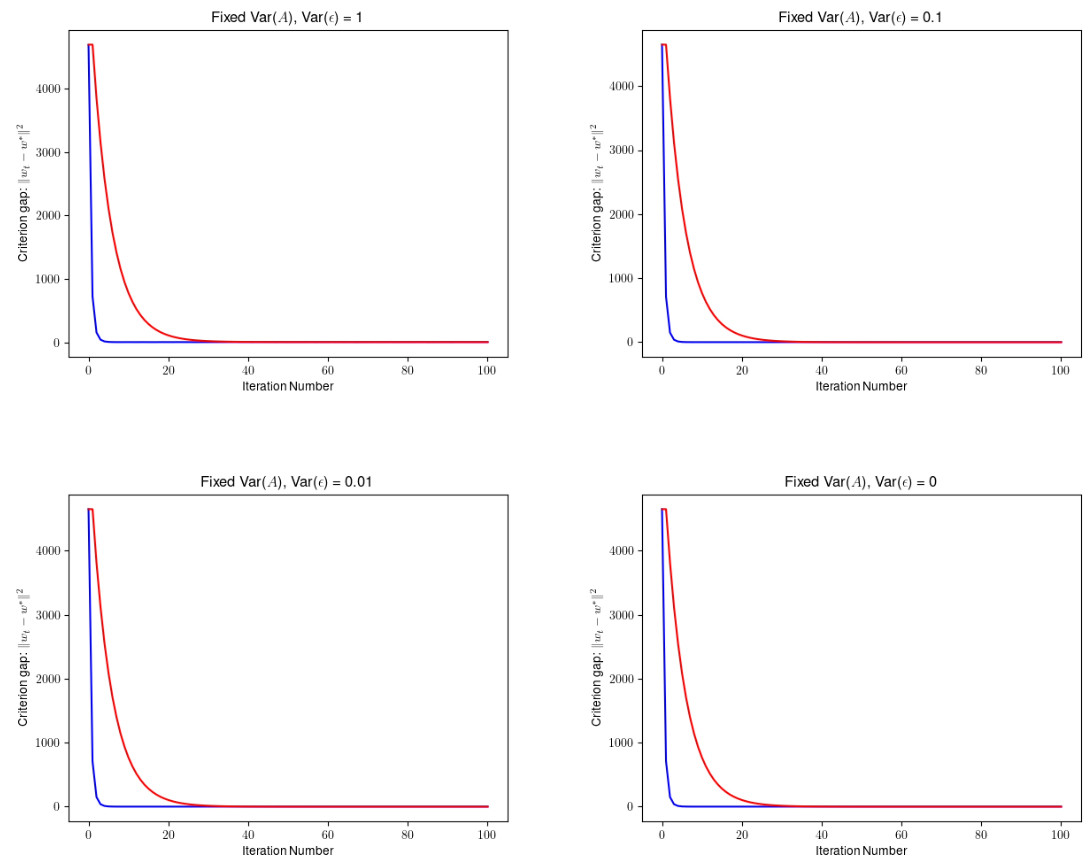
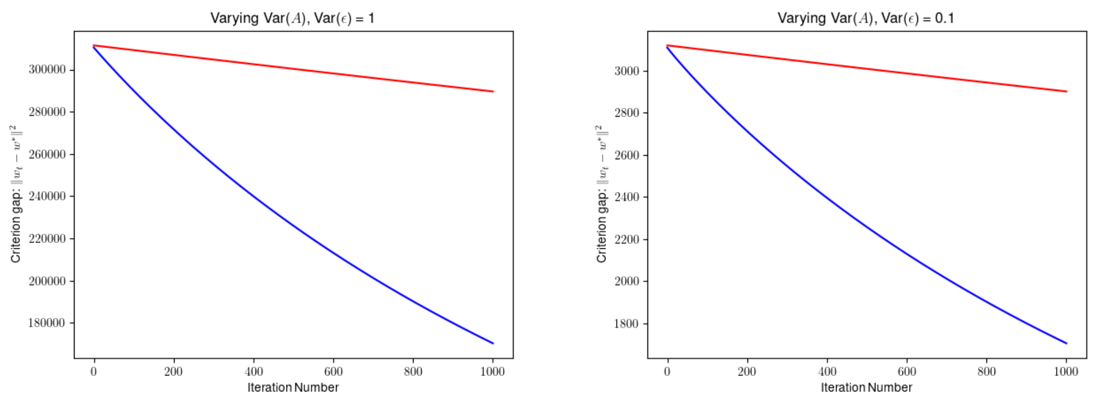


Figure 1: First round of images, theoretical bound in red and empirical result in blue.

Here is the second round of plots where A was chosen to be a $10^4 \times 10^3$ matrix whose jth row had entries sampled from $\mathcal{N}(0, 1/\sqrt{10000j})$, denoted by A_2 below. The batchsize was chosen to be 10, and as the convergence was slower, the algorithm was allowed to run for 1000 iterations with a step size of 0.1.



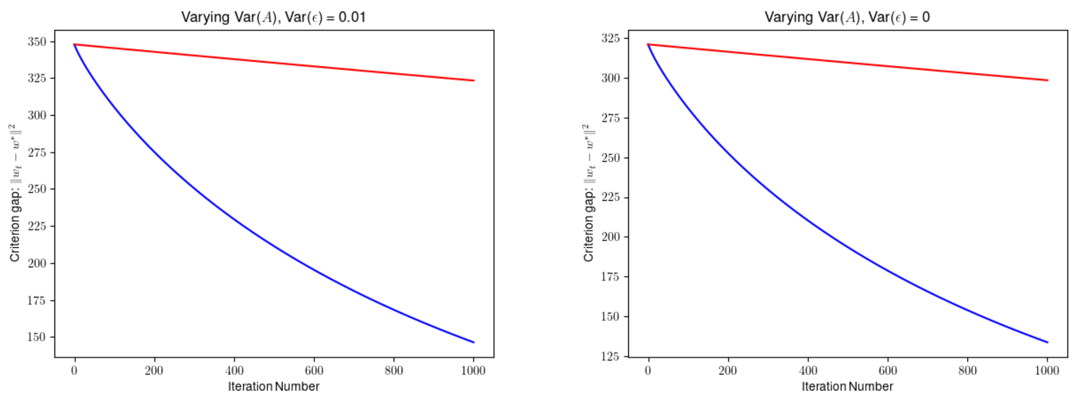


Figure 2: Second round of images, theoretical bound in red and empirical result in blue.

The theoretical result was pretty good in the first round of plots, but it was shockingly loose in the second round. I'm not sure why this is, and to be honest, I don't fully trust my implementation of it. The bound used was the following, taken from Slide 16 of Lecture 18:

$$\mathbb{E}\|w_t - w_*\|_2^2 \leq (1 - 2\alpha\mu(1-\alpha L))^t \|w_0 - w_*\|_2^2 + \alpha\frac{\sigma^2}{\mu(1-\alpha L)},$$

where

- $\mu = \lambda_{min}(A^\top A)$
- $L = n \cdot \max_j \|a_j\|_2^2$ where a_j is the jth row of A
- $\sigma^2 = \frac{1}{n}\sum_{j=1}^n \|\nabla f_j(w_*)\|^2$
- α is the stepsize, reset to $1/2L$ if larger than $1/L$.

I did make one change to this bound, I multiplied t in the exponent by the batchsize. This sped up the convergence, but may not be theoretically correct in general.

Here is a table presenting the results in another way. I believe the error would have continued to fall for the A_2 case, if more iterations were used. I wasn't patient enough to let it run for longer, however.

			Experimental			Theoretical	
A	ν^2	Initial Error	Final Error	Reduction %	Initial Error	Final Error	Reduction
A_1	1	4690.1205	4.6306	99.9013	4696.5921	6.4717	99.8622
A_1	0.1	4647.4501	0.0477	99.9990	4647.5172	0.0669	99.9985
A_1	0.01	4651.7212	0.0004672	99.9999	4651.7219	0.0006495	99.9999
A_1	0	4651.9180	1.5432e-09	99.9999	4651.9180	1.1542	99.9999
A_2	1	310402.7632	170335.4203	45.1243	311385.3539	289588.1955	7.0000
A_2	1	3108.1134	1705.7998	45.1178	3118.6050	2900.3430	6.9986
A_2	1	347.6325	146.3506	57.9008	347.7354	323.3239	7.0201
A_2	1	321.1256	133.5297	58.4182	321.1256	298.5754	7.0222

□

EXERCISE 4. Consider a three-state Markov chain with stationary probabilities $\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right)$. consider the Metropolis-Hastings algorithm with G the complete graph on these three vertices. For each edge and each direction, what is the expected probability that we would actually make a move along the edge?

Proof: Recall that the Metropolis transition probabilities are

$$p_{xy} = \frac{1}{r} \min\left(1, \frac{\pi(y)}{\pi(x)}\right)$$

if x and y are distinct but adjacent and

$$p_{xx} = 1 - \sum_{y \neq x} p_{xy}.$$

Let a, b and c be the vertices of the graph. Then

$$\begin{aligned} p_{ab} &= \frac{1}{2}\frac{2}{1} \cdot \frac{1}{3} = \frac{1}{3} \\ p_{ac} &= \frac{1}{2}\frac{2}{1} \cdot \frac{1}{6} = \frac{1}{6} \\ p_{aa} &= 1 - \frac{1}{3} - \frac{1}{6} = \frac{1}{2}. \end{aligned}$$

The other transition probabilities are

$$p_{ba} = \frac{1}{2}, \qquad p_{bc} = \frac{1}{4}, \qquad p_{bb} = \frac{1}{4}$$

and

$$p_{ca} = \frac{1}{2}, \qquad p_{cb} = \frac{1}{2}, \qquad p_{cc} = 0.$$

□

EXERCISE 5. Consider the probability distribution $p(\mathbf{x})$ where $\mathbf{x} \in \{0,1\}^{100}$ such that $p(0) = \frac{1}{2}$ and $p(\mathbf{x}) = \frac{1/2}{2^{100}-1}$. How does Gibbs sampling behave here?

Proof: The Gibbs transition probabilities are given by

$$p_{xy} = \begin{cases} \frac{1}{d}\pi(y_i \mid x_1, ..., \hat{x}_i, ..., x_d) & \text{if } \mathbf{x} \text{ and } \mathbf{y} \text{ differ only in } i \\ 0 & \text{otherwise} \end{cases}.$$

Let $\hat{e}_i$ denote the element of $\{0,1\}^{100}$ whose ith component is 1 and is 0 elsewhere. We have three cases to examine.

If we are currently at $\mathbf{0}$, then

- there is a $\frac{1}{100}\frac{\frac{1/2}{2^{100}-1}}{\frac{1}{2}+\frac{1/2}{2^{100}-1}} \approx \frac{1}{100} \cdot \frac{1}{2^{100}-1}$ chance of moving to $\hat{e}_i$ for any $i \in \{1, ..., 100\}$. Altogether, we have a 1 in $2^{100} - 1$ chance of leaving 0 at all.

- We have a $1 - \frac{1}{2^{100}-1} \approx 1$ chance of remaining at zero.

Hence, if we ever reach 0 then we will stay at zero, since $2^{100} - 1$ is a huge number.

If we are currently at $\hat{e}_i$, then we

- have a $\frac{1}{100} \frac{1/2}{\frac{1}{2} + \frac{1/2}{2^{100}-1}} \approx \frac{1}{100}$ chance of moving to 0.
- have a $\frac{1}{100} \frac{\frac{1/2}{2^{100}-1}}{\frac{1/2}{2^{100}-1} + \frac{1/2}{2^{100}-1}} \approx \frac{1}{200}$ chance of moving to some other nonzero point, of which there are 99 adjacent to $\hat{e}_i$ giving us an approximately $\frac{1}{2}$ chance point of moving to a point which is not 0
- have an approximately $1 - \frac{1}{100} - \frac{1}{2} = \frac{1}{2} - \frac{1}{100}$ chance of remaining at $\hat{e}_i$.

If we are currently at $\mathbf{x} \neq \hat{e}_i, \mathbf{0}$, then we

- have a $\frac{1}{100} \frac{\frac{1/2}{2^{100}-1}}{\frac{1/2}{2^{100}-1} + \frac{1/2}{2^{100}-1}} \approx \frac{1}{200}$ chance of moving to any individual neighbor of $\mathbf{x}$, or altogether a 1 in 2 chance of leaving $\mathbf{x}$ to *some* other point
- have an $1 - 100 \cdot \frac{1}{200} \approx \frac{1}{2}$ chance of remaining at $\mathbf{x}$.

If we initialize a random walk on G at $\mathbf{0}$ then we will remain there functionally forever. If we initialize it at any other point, then we have a $\frac{1}{2}$ chance to leave and a $\frac{1}{2}$ chance to remain. The situation is slightly different at a point neighboring $\mathbf{0}$, where we have twice the chance of transitioning to $\mathbf{0}$ than to any other point. Thus, a random walk on G will visit a variety of points, transitioning to a new point every 2 steps on average, unless it reaches $\mathbf{0}$, in which case it will remain there indefinitely. However, the chance of reaching $\mathbf{0}$ from a random initialization is just as small as the chance of leaving $\mathbf{0}$, since there are 2^{100} points in total. $\square$

EXERCISE 6. Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a matrix with rank exactly k. Consider the following randomized procedure:

- Draw a matrix $\mathbf{G} \in \mathbb{R}^{n \times k}$ with i.i.d. $\mathcal{N}(0, 1)$ Gaussian entries.
- Form the matrix $\mathbf{Y} = \mathbf{AG} \in \mathbb{R}^{m \times k}$.
- Form the matrix $\mathbf{Z} = \mathbf{A}^\top \mathbf{Y} \in \mathbb{R}^{n \times k}$

Is it possible to reconstruct $\mathbf{A}$ and its singular value decomposition from the information provided by the matrices $(\mathbf{G}, \mathbf{Y}, \mathbf{Z})$? If yes, then specify and algorithm. If no, then provide a counterexample.

Proof: The answer is 'yes'. Perhaps this expected, for we know that $\mathbf{G}$ has full rank k. Indeed, because of this, and because if B has full rank then $\mathrm{rank}(AB) = \mathrm{rank}(A)$, we know that $\mathbf{Y} = \mathbf{AG}$ has rank k. This means $\mathbf{Y}$ has reduced SVD

$$\mathbf{Y} = U_Y \Sigma_Y V_Y^\top.$$

Because $\mathbf{Y}$ is $m \times k$ and is rank k, it is full rank and hence $\mathbf{Z} = \mathbf{A}^\top \mathbf{Y}$ has rank k. Thus $\mathbf{Z}$ has reduced SVD

$U_Z \Sigma_Z V_Z^\top$, giving us

$$\mathbf{Z} = U_Z \Sigma_Z V_Z^\top = \mathbf{A}^\top \mathbf{Y} = \mathbf{A}^\top U_Y \Sigma_Y V_Y^\top.$$

As $\mathbf{Y}$ is of full rank k, we may find a matrix M such that $A^\top \mathbf{Y} C = A^{\mathbf{Y}}$ (hand waving has occurred). Solving for $\mathbf{A}^\top$ using this fact gives us

$$\begin{aligned} A^T &= U_Z \Sigma_Z V_Z (U_Y \Sigma_Y V_Y^\top)^\top C \\ &= U_Z \Sigma_Z V_Z^\top V_Z \Sigma_Y U_Y^\top C \\ &= U_Z \Sigma_Z \Sigma_Y U_Y^\top C \end{aligned}$$

Notice that this is an SVD, $\Sigma_Z \Sigma_Y$ is diagonal. Hence we can recover the SVD of $\mathbf{A}$ (and hence $\mathbf{A}$) by calculating the SVD of $\mathbf{Y}$ and $\mathbf{Z}$ and setting $U_A = C^\top U_Y$, $\Sigma_A = \Sigma_Y \Sigma_Z$ and $V_A = U_Z$. □

```
from multiprocessing import Value
from matplotlib.pyplot import plot
import numpy as np
import math
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split


def sgd(
    gradient,
    gradient_js,
    x,
    y,
    start,
    ideal,
    learn_rate=0.1,
    batch_size=1,
    n_iter=50,
    tolerance=1e-06,
    dtype="float64",
    random_state=None,
    plot_title="",
    mul=1,
    filename="",
):
    # Checking if the gradient is callable
    if not callable(gradient):
        raise TypeError("'gradient' must be callable")

    # Setting up the data type for NumPy arrays
    dtype_ = np.dtype(dtype)

    # Converting x and y to NumPy arrays
    x, y = np.array(x, dtype=dtype_), np.array(y, dtype=dtype_)
    n_obs = x.shape[0]
    if n_obs != y.shape[0]:
        raise ValueError("'x' and 'y' lengths do not match")
    # Make matrix out of all x and y points, smash the matrices together columnwise
    # This makes shuffling easier later on by shuffling x and y simultaneously
    xy = np.c_[x.reshape(n_obs, -1), y.reshape(n_obs, 1)]

    # Initializing the random number generator
    seed = None if random_state is None else int(random_state)
```

```
rng = np.random.default_rng(seed=seed)

# Initializing the values of the variables
vector = np.array(start, dtype=dtype_)

# Setting up and checking the learning rate
learn_rate = np.array(learn_rate, dtype=dtype_)
if np.any(learn_rate <= 0):
    raise ValueError("'learn_rate' must be greater than zero")

# Setting up and checking the size of minibatches
batch_size = int(batch_size)
print("batch_size:", batch_size)
if not 0 < batch_size <= n_obs:
    raise ValueError(
        "'batch_size' must be greater than zero and less than "
        "or equal to the number of observations"
    )

# Setting up and checking the maximal number of iterations
n_iter = int(n_iter)
if n_iter <= 0:
    raise ValueError("'n_tier' must be greater than zero")

# Setting up and checking the tolerance
tolerance = np.array(tolerance, dtype=dtype_)
if np.any(tolerance <= 0):
    raise ValueError("'tolerance' must be greater than zero")

# initialize plotting variables
accuracy = []
theoretical = []
iter_num = []

# deriving constants
# lecture 18 page 16
def terms_in_bound(learn_rate):
    # get eigenvalues, n, and all norms of the rows of x
    eig, _ = np.linalg.eig(np.matmul(x.T, x))
    n = x.shape[0]
    row_norm = [(np.linalg.norm(x[j, :])) ** 2 for j in range(rows)]

    # get sigma
```

```
        inp_diff = [np.abs((np.dot(x[j, :], ideal) - y[j])) ** 2 for j in range(n)]
        sigma2 = (
            n ** (1.4)
            * sum([row_norm[j] * inp_diff[j] for j in range(n)])
            / (batch_size**2)
        )[0]

        # get the constants L, mu, alpha etc
        mu = min(sorted(eig))
        L = n * max(row_norm)

        if learn_rate >= 1 / (L):
            alpha = 1 / (2 * L)

        else:
            alpha = learn_rate
        grad = gradient(x, y, ideal)

        print("alpha: ", alpha)
        print("L: ", L)
        print("mu: ", mu)
        print("sigma^2: ", sigma2)

        # terms in bound
        coeff = 1 - 2 * alpha * mu * (1 - alpha * L)
        norm = np.linalg.norm(start - ideal) ** 2
        frac = alpha * (sigma2) / (mu * (1 - alpha * L))
        print(f"alpha: {alpha}\nalpha/(mu(1 - alpha*L)): {frac/sigma2}")
        return coeff, norm, frac

    def theory_bound(i, coeff, norm, frac):
        return (coeff**i) * norm + frac

    # compute these terms once
    coeff, norm, frac = 0, 0, 0
    if plot_title:
        coeff, norm, frac = terms_in_bound(learn_rate=learn_rate)
        print(coeff, norm, frac)

    accuracy.append(np.linalg.norm(start - ideal) ** 2)
    theoretical.append(theory_bound(0, coeff, norm, frac))
    iter_num.append(0)
    # Performing the gradient descent loop
```

```
for i in range(n_iter):
    # Shuffle x and y
    rng.shuffle(xy)

    # Performing minibatch moves
    for start in range(0, n_obs, batch_size):
        stop = start + batch_size
        x_batch, y_batch = xy[start:stop, :-1], xy[start:stop, -1:]

        # Recalculating the difference
        grad = np.array(gradient(x_batch, y_batch, vector), dtype_)
        diff = -learn_rate * grad
        # Checking if the absolute difference is small enough
        if np.all(np.linalg.norm(diff) <= tolerance):
            break

        # Updating the values of the variables
        vector += diff

    acc = np.linalg.norm(vector - ideal) ** 2
    # print(f"completed iteration {i}, ||pred - b|| = {acc}")
    if plot_title:
        iter_num.append(i + 1)
        accuracy.append(acc)
        theoretical.append(theory_bound(i * mul, coeff, norm, frac))

if plot_title:
    import matplotlib.pyplot as plt

    plt.rcParams.update({"text.usetex": True, "font.family": "Helvetica"})
    plt.plot(iter_num, accuracy, c="b")
    plt.plot(iter_num, theoretical, c="r")
    plt.xlabel("Iteration Number")
    plt.ylabel(r"Criterion gap: $ \|w_t- w^*\|^2 $")
    plt.title(plot_title)
    plt.savefig(filename)
    plt.close()

print("\n\nExperimental, Theoretical")
print(
    accuracy[0],
    "&",
    accuracy[-1],
        "&",
        ((accuracy[0] - accuracy[-1]) / accuracy[0]) * 100,
        "&",
        theoretical[0],
        "&",
        theoretical[-1],
        "&",
        ((theoretical[0] - theoretical[-1]) / theoretical[0]) * 100,
    )
    print("\n")
    return vector if vector.shape else vector.item()


# the rows and columns of the matrices in this problem
rows = 10000
cols = 1000

# the various epsilons
ep1 = np.random.normal(0, 1, size=(rows, 1))
ep2 = np.random.normal(0, 0.1, size=(rows, 1))
ep3 = np.random.normal(0, 0.01, size=(rows, 1))
ep4 = np.random.normal(0, 0, size=(rows, 1))

# a cols x 1 column vector, correct shape to multiply with A
ones = np.ones(shape=(cols, 1))


def problem3a():
    mu = 0
    sigma = 1 / math.sqrt(cols)
    A = np.random.normal(mu, sigma, (rows, cols))

    # setup the b and w vectors
    M = np.matmul(np.linalg.inv(np.matmul(np.transpose(A), A)), np.transpose(A))
    b0 = np.matmul(A, ones)
    bs = [b0 + ep1, b0 + ep2, b0 + ep3, b0 + ep4]
    w_min = [np.matmul(M, b) for b in bs]
    w_0 = np.random.uniform(low=0, high=5, size=(cols, 1))

    # hyper-parameters
    stepsize = 0.1
    batch_size = 100
    num_iter = 100
```

```
mul = 1000

# graphing
titles = [
    r"Fixed Var($A$), Var($\epsilon$) = 1",
    r"Fixed Var($A$), Var($\epsilon$) = 0.1",
    r"Fixed Var($A$), Var($\epsilon$) = 0.01",
    r"Fixed Var($A$), Var($\epsilon$) = 0",
]
filenames = ["graph1.png", "graph2.png", "graph3.png", "graph4.png"]

def grad_loss(X, y, w):
    """
    Gradient of the least squares loss function
    """
    return np.matmul(np.transpose(X), np.matmul(X, w) - y)

def grad_loss_js(X, y, w):
    """
    Gradient of the least squares loss function for the jth part
    """
    AtA = np.matmul(X.transpose(), X)
    n = w.shape[0]
    stupid = [AtA[:, j].reshape((n, 1)) * w[j] for j in range(n)]
    Atb = np.matmul(X.transpose(), y)
    print(AtA)
    print(Atb)
    print()

    components = [stupid[j] + (Atb / n) for j in range(n)]
    return components

for i in range(len(bs)):
    b = bs[i]
    w_best = w_min[i]
    title = titles[i]
    sgd(
        gradient=grad_loss,
        gradient_js=grad_loss_js,
        x=A,
        y=bs[i],
        start=w_0,
        learn_rate=stepsize,
            ideal=w_min[i],
            batch_size=batch_size,
            n_iter=num_iter,
            plot_title=titles[i],
            mul=mul,
            filename=filenames[i],
        )


def problem3b():
    mu = 0
    sigma = lambda j: 1 / math.sqrt(cols * j)
    A = np.array([np.random.normal(mu, sigma(j), (cols)) for j in range(1, rows + 1)])

    # setup the b and w vectors
    M = np.matmul(np.linalg.inv(np.matmul(np.transpose(A), A)), np.transpose(A))
    b0 = np.matmul(A, ones)
    bs = [b0 + ep1, b0 + ep2, b0 + ep3, b0 + ep4]
    w_min = [np.matmul(M, b) for b in bs]
    w_0 = np.random.uniform(0, 1, size=(cols, 1))

    # hyper-parameters
    stepsize = 0.1
    batch_size = 10
    num_iter = 1000
    mul = 1000

    # graphing
    titles = [
        r"Varying Var($A$), Var($\epsilon$) = 1",
        r"Varying Var($A$), Var($\epsilon$) = 0.1",
        r"Varying Var($A$), Var($\epsilon$) = 0.01",
        r"Varying Var($A$), Var($\epsilon$) = 0",
    ]
    filenames = ["graph5.png", "graph6.png", "graph7.png", "graph8.png"]

    def grad_loss(X, y, w):
        """
        Gradient of the least squares loss function
        """
        return np.matmul(np.transpose(X), np.matmul(X, w) - y)

    def grad_loss_js(X, y, w):
        """
        Gradient of the least squares loss function for the jth part
        """
        AtA = np.matmul(X.transpose(), X)
        n = w.shape[0]
        stupid = [AtA[:, j].reshape((n, 1)) * w[j] for j in range(n)]
        Atb = np.matmul(X.transpose(), y)
        print(AtA)
        print(Atb)
        print()

        components = [stupid[j] + (Atb / n) for j in range(n)]
        return components

    for i in range(len(bs)):
        b = bs[i]
        w_best = w_min[i]
        title = titles[i]
        sgd(
            gradient=grad_loss,
            gradient_js=grad_loss_js,
            x=A,
            y=bs[i],
            start=w_0,
            learn_rate=stepsize,
            ideal=w_min[i],
            batch_size=batch_size,
            n_iter=num_iter,
            plot_title=titles[i],
            mul=mul,
            filename=filenames[i],
        )


problem3b()
```