```
# Useful starting lines
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
%reload_ext autoreload
%autoreload 2
```

```
# Check the Python version
import sys
if sys.version.startswith("3."):
  print("You are running Python 3. Good job :)")
else:
  print("This notebook requires Python 3.\nIf you are using Google Colab, go to Runtime > Change runtime type and choose Python 3.")
```

```
You are running Python 3. Good job :)
```

```
# Load the data
```

```
import datetime
from helpers import *

height, weight, gender = load_data(sub_sample=False, add_outlier=False)
x, mean_x, std_x = standardize(height)
b, A = build_model_data(x, weight)
```

```
print('Number of samples n = ', b.shape[0])
print('Dimension of each sample d = ', A.shape[1])
```

```
Number of samples n =  10000
Dimension of each sample d =  2
```

Least Squares Estimation

Least squares estimation is one of the fundamental machine learning algorithms. Given an $ n \times d $ matrix A and a $ n \times 1$ vector b, the goal is to find a vector $x \in \mathbb{R}^d$

which minimizes the objective function f(x) = \frac{1}{2n} \sum_{i=1}^{n} (a_i^\top x - b_i)^2 = \frac{1}{2n} |Ax - b|^2

In this exercise, we will try to fit x using Least Squares Estimation.

One can see the function is L smooth with $L =\frac1n|A^T A| = \frac1n|A|^2$.

Computing the Objective Function

Fill in the `calculate_objective` function below:

```
def calculate_objective(Axmb):
    """Calculate the mean squared error for vector Axmb = Ax - b."""
    # ************************************************
    # INSERT YOUR CODE HERE
    # TODO: compute mean squared error
    # ************************************************
    squared = Axmb ** 2
    sum_squared = np.sum(squared)
    mse = 0.5 * sum_squared / len(Axmb)
    return mse
```

Compute smoothness constant L

To compute the spectral norm of A you can use np.linalg.norm(A, 2)

```
def calculate_L(b, A):
    """Calculate the smoothness constant for f"""
    # ************************************************
    # INSERT YOUR CODE HERE
    # TODO: compute ||A.T*A||
    # ************************************************
    ATA = A.T @ A
    # ************************************************
    # INSERT YOUR CODE HERE
    # TODO: compute L = smoothness constant of f
    # ************************************************
    L = np.linalg,norm(ATA, 2) / len(Axmb)
    return L
```

Gradient Descent

Please fill in the functions `compute_gradient` below:

```
def compute_gradient(b, A, x):
    """Compute the gradient."""
    # ***************************************************
    # INSERT YOUR CODE HERE
    # TODO: compute gradient and objective
    # ***************************************************
    Axmb = A @ x - b
    grad = A.T @ Axmb / len(Axmb)
    return grad, Axmb
```

Please fill in the functions `gradient_descent` below:

```
def gradient_descent(b, A, initial_x, max_iters, gamma):
    """Gradient descent algorithm."""
    # Define parameters to store x and objective func. values
    xs = [initial_x]
    objectives = []
    x = initial_x
    for n_iter in range(max_iters):
        # ***************************************************
        # INSERT YOUR CODE HERE
        # TODO: compute gradient and objective function
        # ***************************************************
        grad, Axmb = compute_gradient(b, A, x)
        obj = 0.5 * np.sum(Axmb ** 2) / len (Axmb)
        # ***************************************************
        # INSERT YOUR CODE HERE
        # TODO: update x by a gradient descent step
        # ***************************************************
        x = x - gamma * grad
        # store x and objective function value
        xs.append(x)
        objectives.append(obj)
        print("Gradient Descent({bi}/{ti}): objective={l}".format(
              bi=n_iter, ti=max_iters - 1, l=obj))

    return objectives, xs
```

Test your gradient descent function with a naive step size through gradient descent demo shown below:

```
# from gradient_descent import *
from plots import gradient_descent_visualization
```

```
3.
4. # Define the parameters of the algorithm.
5. max_iters = 50
6.
7. gamma = 0.85
8.
9. # Initialization
10. x_initial = np.zeros(A.shape[1])
11.
12. # Start gradient descent.
13. start_time = datetime.datetime.now()
14. gradient_objectives_naive, gradient_xs_naive = gradient_descent(b, A, x_initial,
    max_iters, gamma)
15. end_time = datetime.datetime.now()
16.
17. # Print result
18. exection_time = (end_time - start_time).total_seconds()
19. print("Gradient Descent: execution time={t:.3f} seconds".format(t=exection_time))
```

```
1. Gradient Descent(0/49): objective=2792.2367127591674
2. Gradient Descent(1/49): objective=77.86503142886588
3. Gradient Descent(2/49): objective=16.791668598930205
4. Gradient Descent(3/49): objective=15.417517935256669
5. Gradient Descent(4/49): objective=15.386599545324014
6. Gradient Descent(5/49): objective=15.385903881550528
7. Gradient Descent(6/49): objective=15.385888229115624
8. Gradient Descent(7/49): objective=15.385887876935838
9. Gradient Descent(8/49): objective=15.385887869011794
10. Gradient Descent(9/49): objective=15.385887868833503
11. Gradient Descent(10/49): objective=15.385887868829494
12. Gradient Descent(11/49): objective=15.385887868829403
13. Gradient Descent(12/49): objective=15.3858878688294
14. Gradient Descent(13/49): objective=15.385887868829398
15. Gradient Descent(14/49): objective=15.3858878688294
16. Gradient Descent(15/49): objective=15.3858878688294
17. Gradient Descent(16/49): objective=15.385887868829398
18. Gradient Descent(17/49): objective=15.3858878688294
19. Gradient Descent(18/49): objective=15.3858878688294
20. Gradient Descent(19/49): objective=15.3858878688294
21. Gradient Descent(20/49): objective=15.3858878688294
22. Gradient Descent(21/49): objective=15.3858878688294
23. Gradient Descent(22/49): objective=15.3858878688294
24. Gradient Descent(23/49): objective=15.3858878688294
25. Gradient Descent(24/49): objective=15.3858878688294
26. Gradient Descent(25/49): objective=15.3858878688294
27. Gradient Descent(26/49): objective=15.3858878688294
```

```
Gradient Descent(27/49): objective=15.3858878688294
Gradient Descent(28/49): objective=15.3858878688294
Gradient Descent(29/49): objective=15.3858878688294
Gradient Descent(30/49): objective=15.3858878688294
Gradient Descent(31/49): objective=15.3858878688294
Gradient Descent(32/49): objective=15.3858878688294
Gradient Descent(33/49): objective=15.3858878688294
Gradient Descent(34/49): objective=15.3858878688294
Gradient Descent(35/49): objective=15.3858878688294
Gradient Descent(36/49): objective=15.3858878688294
Gradient Descent(37/49): objective=15.3858878688294
Gradient Descent(38/49): objective=15.3858878688294
Gradient Descent(39/49): objective=15.3858878688294
Gradient Descent(40/49): objective=15.3858878688294
Gradient Descent(41/49): objective=15.3858878688294
Gradient Descent(42/49): objective=15.3858878688294
Gradient Descent(43/49): objective=15.3858878688294
Gradient Descent(44/49): objective=15.3858878688294
Gradient Descent(45/49): objective=15.3858878688294
Gradient Descent(46/49): objective=15.3858878688294
Gradient Descent(47/49): objective=15.3858878688294
Gradient Descent(48/49): objective=15.3858878688294
Gradient Descent(49/49): objective=15.3858878688294
Gradient Descent: execution time=0.002 seconds
```

Time Visualization

```
from ipywidgets import interact, IntSlider
from grid_search import *


def plot_figure(n_iter):
    # Generate grid data for visualization (parameters to be swept and best combination)
    grid_x0, grid_x1 = generate_w(num_intervals=10)
    grid_objectives = grid_search(b, A, grid_x0, grid_x1)
    obj_star, x0_star, x1_star = get_best_parameters(grid_x0, grid_x1, grid_objectives)

    fig = gradient_descent_visualization(
        gradient_objectives, gradient_xs, grid_objectives, grid_x0, grid_x1, mean_x, std_x, height, weight, n_iter)
    fig.set_size_inches(20.0, 4.0)
    display(fig)

```

```
interact(plot_figure, n_iter=IntSlider(min=1, max=len(gradient_xs_naive)))
```

```
interactive(children=(IntSlider(value=1, description='n_iter', max=51, min=1), Output()), _dom_classes=('widge…


<function __main__.plot_figure(n_iter)>
```

Try doing gradient descent with a better learning rate

```
# Define the parameters of the algorithm.
max_iters = 50

# ************************************************
# INSERT YOUR CODE HERE
# TODO: a better learning rate using the smoothness of f
# ************************************************
L = np.linalg.norm(A.T @ A, 2)
gamma = 100 / L

# Initialization
x_initial = np.zeros(A.shape[1])

# Start gradient descent.
start_time = datetime.datetime.now()
gradient_objectives, gradient_xs = gradient_descent(b, A, x_initial, max_iters, gamma)
end_time = datetime.datetime.now()

# Print result
exection_time = (end_time - start_time).total_seconds()
print("Gradient Descent: execution time={t:.3f} seconds".format(t=exection_time))
```

```
Gradient Descent(0/49): objective=780.8686016504854
Gradient Descent(1/49): objective=773.251154488717
Gradient Descent(2/49): objective=766.3890713588925
Gradient Descent(3/49): objective=760.2039370530574
Gradient Descent(4/49): objective=754.6257563657889
Gradient Descent(5/49): objective=749.5920220920866
Gradient Descent(6/49): objective=745.046889499426
Gradient Descent(7/49): objective=740.9404446521098
Gradient Descent(8/49): objective=737.2280555322218
```

```
Gradient Descent(9/49): objective=733.8697962622064
Gradient Descent(10/49): objective=730.8299359180801
Gradient Descent(11/49): objective=728.0764844539572
Gradient Descent(12/49): objective=725.580789158749
Gradient Descent(13/49): objective=723.317175852385
Gradient Descent(14/49): objective=721.2626297169031
Gradient Descent(15/49): objective=719.3965112603233
Gradient Descent(16/49): objective=717.7003034395566
Gradient Descent(17/49): objective=716.1573864323553
Gradient Descent(18/49): objective=714.7528369557789
Gradient Descent(19/49): objective=713.47324938702
Gradient Descent(20/49): objective=712.3065762579041
Gradient Descent(21/49): objective=711.241985972315
Gradient Descent(22/49): objective=710.2697358408802
Gradient Descent(23/49): objective=709.3810587435117
Gradient Descent(24/49): objective=708.5680619213746
Gradient Descent(25/49): objective=707.8236365686215
Gradient Descent(26/49): objective=707.1413770434442
Gradient Descent(27/49): objective=706.5155086500336
Gradient Descent(28/49): objective=705.9408230599074
Gradient Descent(29/49): objective=705.4126205446067
Gradient Descent(30/49): objective=704.9266582834891
Gradient Descent(31/49): objective=704.4791040917067
Gradient Descent(32/49): objective=704.0664949855989
Gradient Descent(33/49): objective=703.6857000667632
Gradient Descent(34/49): objective=703.333887262911
Gradient Descent(35/49): objective=703.0084935140999
Gradient Descent(36/49): objective=702.7071980377789
Gradient Descent(37/49): objective=702.42789834596
Gradient Descent(38/49): objective=702.1686887232516
Gradient Descent(39/49): objective=701.927840906029
Gradient Descent(40/49): objective=701.703786731037
Gradient Descent(41/49): objective=701.495102546697
Gradient Descent(42/49): objective=701.3004952025822
Gradient Descent(43/49): objective=701.1187894523173
Gradient Descent(44/49): objective=700.9489166227798
Gradient Descent(45/49): objective=700.7899044181594
Gradient Descent(46/49): objective=700.6408677414381
Gradient Descent(47/49): objective=700.5010004283125
Gradient Descent(48/49): objective=700.3695677996996
Gradient Descent(49/49): objective=700.2458999488812
Gradient Descent: execution time=0.001 seconds
```

Time visualization with a better learning rate

```
def plot_figure(n_iter):
```

```
    # Generate grid data for visualization (parameters to be swept and best combination)
    grid_x0, grid_x1 = generate_w(num_intervals=10)
    grid_objectives = grid_search(b, A, grid_x0, grid_x1)
    obj_star, x0_star, x1_star = get_best_parameters(grid_x0, grid_x1, grid_objectives)

    fig = gradient_descent_visualization(
        gradient_objectives, gradient_xs, grid_objectives, grid_x0, grid_x1, mean_x, std_x, height, weight, n_iter)
    fig.set_size_inches(10.0, 6.0)
    display(fig)

interact(plot_figure, n_iter=IntSlider(min=1, max=len(gradient_xs)))
```

```
interactive(children=(IntSlider(value=1, description='n_iter', max=51, min=1), Output()), _dom_classes=('widge…



<function __main__.plot_figure(n_iter)>
```

Loading more complex data

The data is taken from https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength

```
data = np.loadtxt("Concrete_Data.csv",delimiter=",")

A = data[:,:-1]
b = data[:,-1]
A, mean_A, std_A = standardize(A)
```

```
print('Number of samples n = ', b.shape[0])
print('Dimension of each sample d = ', A.shape[1])
```

```
Number of samples n =  1030
Dimension of each sample d =  8
```

Running gradient descent

Assuming bounded gradients

Assume we are moving in a bounded region $|x| \leq 25$ containing all iterates (and we assume $|x-x^\star| \leq 25$ as well, for simplicity). Then by $\nabla f(x) = \frac{1}{n}A^\top (Ax - b)$, one can see that f is Lipschitz over that bounded region, with Lipschitz constant $|\nabla f(x)| \leq \frac{1}{n} (|A^\top A||x| + |A^\top b|)$

```
# ****************************************************
# INSERT YOUR CODE HERE
# TODO: Compute the bound on the gradient norm
# ****************************************************
norm_ATA = np.linalg.norm(A.T @ A, 2)
norm_ATb = np.linalg.norm(A.T @ b, 2)
n = b.shape[0]
grad_norm_bound = (norm_ATA * 25 + norm_ATb) / n
```

Fill in the learning rate assuming bounded gradients

```
max_iters = 50

# ****************************************************
# INSERT YOUR CODE HERE
# TODO: Compute learning rate based on bounded gradient
# ****************************************************

gamma = 10 / grad_norm_bound
print("grad_norm_bound:", grad_norm_bound)
print("gamma:", gamma)

# Initialization
x_initial = np.zeros(A.shape[1])

# Start gradient descent.
start_time = datetime.datetime.now()
bd_gradient_objectives, bd_gradient_xs = gradient_descent(b, A, x_initial, max_iters, gamma)
end_time = datetime.datetime.now()


# Print result
exection_time = (end_time - start_time).total_seconds()
print("Gradient Descent: execution time={t:.3f} seconds".format(t=exection_time))
```

```
24. 
25. # Averaging the iterates as is the case for bounded gradients case
26. bd_gradient_objectives_averaged = []
27. for i in range(len(bd_gradient_xs)):
28.     if i > 0:
29.         bd_gradient_xs[i] = (i * bd_gradient_xs[i-1] + bd_gradient_xs[i])/(i + 1)
30.     grad, err = compute_gradient(b, A, bd_gradient_xs[i])
31.     obj = calculate_objective(err)
32.     bd_gradient_objectives_averaged.append(obj)
```

```
1. grad_norm_bound: 70.55157320349318
2. gamma: 0.14174028368094385
3. Gradient Descent(0/49): objective=780.8686016504854
4. Gradient Descent(1/49): objective=757.0572465196063
5. Gradient Descent(2/49): objective=740.5812543624813
6. Gradient Descent(3/49): objective=729.084584479613
7. Gradient Descent(4/49): objective=720.9907413148369
8. Gradient Descent(5/49): objective=715.2382349171139
9. Gradient Descent(6/49): objective=711.1080185774268
10. Gradient Descent(7/49): objective=708.1100789289962
11. Gradient Descent(8/49): objective=705.9084315994309
12. Gradient Descent(9/49): objective=704.2712418678465
13. Gradient Descent(10/49): objective=703.0374846206412
14. Gradient Descent(11/49): objective=702.0945497450593
15. Gradient Descent(12/49): objective=701.3631255793681
16. Gradient Descent(13/49): objective=700.7869435574245
17. Gradient Descent(14/49): objective=700.3257840958671
18. Gradient Descent(15/49): objective=699.9506801797878
19. Gradient Descent(16/49): objective=699.6406088594872
20. Gradient Descent(17/49): objective=699.3801950738394
21. Gradient Descent(18/49): objective=699.1581078299688
22. Gradient Descent(19/49): objective=698.9659325325084
23. Gradient Descent(20/49): objective=698.7973726954689
24. Gradient Descent(21/49): objective=698.6476809154398
25. Gradient Descent(22/49): objective=698.5132504450324
26. Gradient Descent(23/49): objective=698.3913200155959
27. Gradient Descent(24/49): objective=698.2797590603523
28. Gradient Descent(25/49): objective=698.1769104076669
29. Gradient Descent(26/49): objective=698.0814743343121
30. Gradient Descent(27/49): objective=697.992422585014
31. Gradient Descent(28/49): objective=697.9089342458274
32. Gradient Descent(29/49): objective=697.8303476560615
33. Gradient Descent(30/49): objective=697.7561241621568
34. Gradient Descent(31/49): objective=697.6858206650937
35. Gradient Descent(32/49): objective=697.6190687328028
36. Gradient Descent(33/49): objective=697.5555586384422
```

```
Gradient Descent(34/49): objective=697.495027111916
Gradient Descent(35/49): objective=697.4372479026222
Gradient Descent(36/49): objective=697.3820244790351
Gradient Descent(37/49): objective=697.329184358537
Gradient Descent(38/49): objective=697.2785746852895
Gradient Descent(39/49): objective=697.2300587666316
Gradient Descent(40/49): objective=697.183513347901
Gradient Descent(41/49): objective=697.1388264577861
Gradient Descent(42/49): objective=697.0958956957468
Gradient Descent(43/49): objective=697.0546268629419
Gradient Descent(44/49): objective=697.0149328608449
Gradient Descent(45/49): objective=696.9767327990826
Gradient Descent(46/49): objective=696.9399512673167
Gradient Descent(47/49): objective=696.9045177361759
Gradient Descent(48/49): objective=696.8703660600796
Gradient Descent(49/49): objective=696.8374340608425
Gradient Descent: execution time=0.002 seconds
```

Gradient descent using smoothness

Fill in the learning rate using smoothness of the function

```
max_iters = 50


# ***************************************************
# INSERT YOUR CODE HERE
# TODO: a better learning rate using the smoothness of f
# ***************************************************
L = np.linalg.norm(A.T @ A, 2)
gamma = 100 / L

# Initialization
x_initial = np.zeros(A.shape[1])

# Start gradient descent.
start_time = datetime.datetime.now()
gradient_objectives, gradient_xs = gradient_descent(b, A, x_initial, max_iters,
gamma)
end_time = datetime.datetime.now()

# Print result
exection_time = (end_time - start_time).total_seconds()
print("Gradient Descent: execution time={t:.3f} seconds".format(t=exection_time))
```

```
Gradient Descent(0/49): objective=780.8686016504854
Gradient Descent(1/49): objective=773.251154488717
Gradient Descent(2/49): objective=766.3890713588925
Gradient Descent(3/49): objective=760.2039370530574
Gradient Descent(4/49): objective=754.6257563657889
Gradient Descent(5/49): objective=749.5920220920866
Gradient Descent(6/49): objective=745.046889499426
Gradient Descent(7/49): objective=740.9404446521098
Gradient Descent(8/49): objective=737.2280555322218
Gradient Descent(9/49): objective=733.8697962622064
Gradient Descent(10/49): objective=730.8299359180801
Gradient Descent(11/49): objective=728.0764844539572
Gradient Descent(12/49): objective=725.580789158749
Gradient Descent(13/49): objective=723.317175852385
Gradient Descent(14/49): objective=721.2626297169031
Gradient Descent(15/49): objective=719.3965112603233
Gradient Descent(16/49): objective=717.7003034395566
Gradient Descent(17/49): objective=716.1573864323553
Gradient Descent(18/49): objective=714.7528369557789
Gradient Descent(19/49): objective=713.47324938702
Gradient Descent(20/49): objective=712.3065762579041
Gradient Descent(21/49): objective=711.241985972315
Gradient Descent(22/49): objective=710.2697358408802
Gradient Descent(23/49): objective=709.3810587435117
Gradient Descent(24/49): objective=708.5680619213746
Gradient Descent(25/49): objective=707.8236365686215
Gradient Descent(26/49): objective=707.1413770434442
Gradient Descent(27/49): objective=706.5155086500336
Gradient Descent(28/49): objective=705.9408230599074
Gradient Descent(29/49): objective=705.4126205446067
Gradient Descent(30/49): objective=704.9266582834891
Gradient Descent(31/49): objective=704.4791040917067
Gradient Descent(32/49): objective=704.0664949855989
Gradient Descent(33/49): objective=703.6857000667632
Gradient Descent(34/49): objective=703.333887262911
Gradient Descent(35/49): objective=703.0084935140999
Gradient Descent(36/49): objective=702.7071980377789
Gradient Descent(37/49): objective=702.42789834596
Gradient Descent(38/49): objective=702.1686887232516
Gradient Descent(39/49): objective=701.927840906029
Gradient Descent(40/49): objective=701.703786731037
Gradient Descent(41/49): objective=701.495102546697
Gradient Descent(42/49): objective=701.3004952025822
Gradient Descent(43/49): objective=701.1187894523173
Gradient Descent(44/49): objective=700.9489166227798
Gradient Descent(45/49): objective=700.7899044181594
```

```
Gradient Descent(46/49): objective=700.6408677414381
Gradient Descent(47/49): objective=700.5010004283125
Gradient Descent(48/49): objective=700.3695677996996
Gradient Descent(49/49): objective=700.2458999488812
Gradient Descent: execution time=0.001 seconds
```

Plotting the Evolution of the Objective Function

```
plt.figure(figsize=(8, 8))
plt.xlabel('Number of steps')
plt.ylabel('Objective Function')
#plt.yscale("log")
plt.plot(range(len(gradient_objectives)), gradient_objectives,'r',
label='gradient descent with 1/L stepsize')
plt.plot(range(len(bd_gradient_objectives)), bd_gradient_objectives,'b',
label='gradient descent assuming bounded gradients')
plt.plot(range(len(bd_gradient_objectives_averaged)),
bd_gradient_objectives_averaged,'g', label='gradient descent assuming bounded
gradients with averaged iterates')
plt.legend(loc='upper right')
plt.show()
```