Gradient Descent

Overview

In this notebook, we will solve a simple linear regression problem by gradient descent.
We will see the effect of the learning rate on the trajectory in parameter space. We will show how Stochastic Gradient Descent (SGD) differs from the standard version, and the effect of "shuffling" your data during SGD.

In [3]:
```python
# Preliminaries - packages to load
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Generate Data from a known distribution

Below we will generate data a known distribution.
Specifically, the true model is:

$Y = b + \theta_1 X_1 + \theta_2 X_2 + \epsilon$

X_1 and X_2 have a uniform distribution on the interval $[0, 10]$, while `const` is a vector of ones (representing the intercept term).

We set actual values for b ,θ_1, and θ_2

Here $b = 1.5$, $\theta_1 = 2$, and $\theta_2 = 5$

We then generate a vector of y-values according to the model and put the predictors together in a "feature matrix" `x_mat`

In [8]:
```python
np.random.seed(1234)  ## This ensures we get the same data if all of the other parameters remain fixed

num_obs = 100
x1 = np.random.uniform(0,10,num_obs) # values btwn 0 and 10 and we want 100 obs
x2 = np.random.uniform(0,10,num_obs)
const = np.ones(num_obs) # consant creates a array of 1's
eps = np.random.normal(0,.5,num_obs) # epsilon(error term) with mean of 0 and a standard deviation of 0.5,

b = 1.5
theta_1 = 2
theta_2 = 5

#Y=b+θ1X1+θ2X2+ϵ

y = b*const+ theta_1*x1 + theta_2*x2 + eps

x_mat = np.array([const,x1,x2]).T
x_mat
```

Out[8]:
```
array([[1.        , 1.9151945 , 7.67116628],
       [1.        , 6.22108771, 7.08115362],
       [1.        , 4.37727739, 7.96867184],
       [1.        , 7.85358584, 5.57760828],
       [1.        , 7.79975808, 9.65836532],
       [1.        , 2.72592605, 1.471569  ],
       [1.        , 2.76464255, 0.29647001],
       [1.        , 8.01872178, 5.93893493],
       [1.        , 9.58139354, 1.14065699],
       [1.        , 8.75932635, 9.5080985 ],
       [1.        , 3.5781727 , 3.25707414],
       [1.        , 5.00995126, 1.9361869 ],
       [1.        , 6.83462935, 4.57811649],
       [1.        , 7.12702027, 9.20402571],
       [1.        , 3.70250755, 8.79069162],
       [1.        , 5.61196186, 2.52615755],
       [1.        , 5.03083165, 3.48008793],
       [1.        , 0.1376845 , 1.82588732],
       [1.        , 7.72826622, 9.01796051],
       [1.        , 8.82641191, 7.06528163],
       [1.        , 3.64885984, 7.26658462],
       [1.        , 6.15396178, 9.00087837],
       [1.        , 0.75381242, 7.79163801],
       [1.        , 3.68824006, 5.99154781],
       [1.        , 9.33140102, 2.91125245],
       [1.        , 6.51378143, 1.51395264],
       [1.        , 3.97202578, 3.35174659],
       [1.        , 7.88730143, 6.57551777],
       [1.        , 3.16836122, 0.73342544],
       [1.        , 5.68098653, 0.55006395],
       [1.        , 8.6912739 , 3.23194814],
       [1.        , 4.36173424, 5.90481804],
       [1.        , 8.02147642, 8.53898567],
       [1.        , 1.43766825, 2.87062425],
       [1.        , 7.04260971, 1.73067227],
       [1.        , 7.04581308, 1.34021206],
       [1.        , 2.18792106, 9.94653829],
       [1.        , 9.24867629, 1.79497869],
       [1.        , 4.42140755, 3.17546823],
       [1.        , 9.09315959, 5.68291405],
       [1.        , 0.59809223, 0.09348575],
       [1.        , 1.84287084, 9.00648621],
       [1.        , 0.47355279, 9.77241431],
       [1.        , 6.74880944, 5.56894679],
       [1.        , 5.9462478 , 0.84773843],
       [1.        , 5.33310163, 3.33002466],
       [1.        , 0.43324063, 7.28428676],
       [1.        , 5.6143308 , 1.42435373],
       [1.        , 3.29668446, 5.52468939],
       [1.        , 5.02966833, 2.7304326 ],
       [1.        , 1.11894318, 9.74495138],
       [1.        , 6.07193706, 6.67786906],
       [1.        , 5.65944643, 2.55653286],
       [1.        , 0.06764062, 1.08311494],
       [1.        , 6.17441709, 7.76180723],
       [1.        , 9.12122886, 7.82477993],
       [1.        , 7.90524133, 7.61603914],
       [1.        , 9.92081466, 9.14403113],
       [1.        , 9.58801762, 6.58622782],
       [1.        , 7.91964135, 5.68367582],
       [1.        , 2.8525096 , 2.01755692],
       [1.        , 6.24916705, 6.98296376],
       [1.        , 4.78093796, 9.5219541 ],
       [1.        , 1.95675179, 8.89963287],
       [1.        , 3.82317452, 9.93567363],
       [1.        , 0.53873685, 8.1870351 ],
       [1.        , 4.51648408, 5.45122166],
       [1.        , 9.82004742, 4.51254055],
       [1.        , 1.239427  , 8.90557188],
       [1.        , 1.19380898, 9.73264791],
       [1.        , 7.38523056, 5.9341133 ],
       [1.        , 5.87303633, 3.66074498],
       [1.        , 4.71632534, 3.23094693],
       [1.        , 1.07126817, 8.71423255],
       [1.        , 2.29218565, 2.15634063],
       [1.        , 8.99965195, 7.34945189],
       [1.        , 4.16753538, 3.65619087],
       [1.        , 5.35851663, 8.01602599],
       [1.        , 0.06208517, 7.82735592],
       [1.        , 3.00641706, 7.01355379],
       [1.        , 4.36893172, 6.22776587],
       [1.        , 6.12148997, 4.93682646],
       [1.        , 9.18198075, 8.405377  ],
       [1.        , 6.2573667 , 7.12096987],
       [1.        , 7.05997565, 4.43908981],
       [1.        , 1.49833716, 0.31034861],
       [1.        , 7.46063409, 3.6323976 ],
       [1.        , 8.31006992, 7.30721791],
       [1.        , 6.33725769, 4.75566573],
       [1.        , 4.38309881, 3.4441697 ],
       [1.        , 1.52572775, 6.40880435],
       [1.        , 5.68409615, 1.26205322],
       [1.        , 5.28224278, 1.71465261],
       [1.        , 9.51428764, 7.37086494],
       [1.        , 4.80359179, 1.27029394],
       [1.        , 5.02559563, 3.69649875],
       [1.        , 5.36878193, 6.04334005],
       [1.        , 8.19202067, 1.03104439],
       [1.        , 0.57115638, 8.02374182],
       [1.        , 6.69421743, 9.45553236]])
```

In [9]:
```python
y
```

Out[9]:
```
array([44.00060834, 49.44119069, 50.57415314, 45.58928189, 65.35503861,
       14.03439564,  8.04255882, 46.6125824 , 26.43591365, 66.44763571,
       26.00356206, 21.26197374, 37.35512528, 62.48566207, 51.78454565,
       24.68094522, 29.14388523, 10.89742951, 62.68253254, 53.75444867,
       44.53288089, 58.51638392, 41.75856245, 38.12132178, 34.82376168,
       21.80088308, 25.46622631, 49.70390141, 12.05602541, 15.39651807,
       34.96172003, 40.19213745, 60.38206962, 18.20268827, 24.07880006,
       21.98268992, 55.68703273, 28.68651837, 26.74897285, 47.70514501,
        2.90129951, 50.25411175, 52.26455684, 43.23633517, 17.88772884,
       28.54311841, 39.3098874 , 20.90432284, 36.44677963, 25.71920238,
       52.83723556, 46.69545869, 25.82169038,  7.39534188, 52.51954727,
       59.82862402, 55.59628039, 67.50716747, 53.72035595, 44.71835284,
       17.09886071, 48.8695994 , 59.23483932, 50.03522379, 58.88530309,
       43.66214118, 37.71052691, 43.33256307, 47.88288694, 52.67558512,
       46.23156426, 32.9317196 , 27.28704805, 47.54794318, 16.72818759,
       56.49680477, 28.54755755, 51.77134925, 40.06492289, 43.1575642 ,
       41.96766483, 38.62279795, 61.45032297, 49.76712273, 38.7473009 ,
        5.19228041, 33.87971392, 54.71962001, 37.95472396, 26.852549  ,
       36.17605578, 19.45541901, 20.38572724, 56.98867987, 18.22335359,
       30.13641244, 42.61077066, 23.47252364, 42.91055714, 62.70436698])
```

Get the "Right" answer directly

In the below cells we solve for the optimal set of coefficients. Note that even though the true model is given by:

$b = 1.5$, $\theta_1 = 2$, and $\theta_2 = 5$

The maximum likelihood (least-squares) estimate from a finite data set may be slightly different.

We can solve the problem two ways:

1. By using the scikit-learn LinearRegression model
2. Using matrix algebra directly via the formula $\theta = (X^T X)^{-1} X^T y$

Note: The scikit-learn solver may give a warning message, this can be ignored.

In [11]:
```python
### Solve directly using sklearn
from sklearn.linear_model import LinearRegression

lr_model = LinearRegression(fit_intercept=False) #intercepts are included in our feature values in our x_mat that we defined earlier.
lr_model.fit(x_mat, y)

lr_model.coef_
```

Out[11]:
```
array([1.49004618, 1.99675416, 5.01156315])
```

b=1 ,theta1 = 2 and theta2 = 5

In [12]:
```python
## Solve by matrix calculation

#θ=(XTX)−1 XTy

#So that's  going to be the inverse of the dot product of the transpose and the value itself,
#and then the dot product of that with x transpose and then the dot product of that with y.

np.linalg.inv(np.dot(x_mat.T,x_mat)).dot(x_mat.T).dot(y)
```

Out[12]:
```
array([1.49004618, 1.99675416, 5.01156315])
```

Same values as with scikit-learn

Solving by Gradient Descent

For most numerical problems, we don't / can't know the underlying analytical solution. This is because we only arrive at analytical solutions by solving the equations mathematically, with pen and paper. That is more often than not just impossible. Fortunately, we have a way of converging to an approximate solution, by using **Gradient Descent**.

lets explore this very useful method because Neural Networks, along with many other complicated algorithms, are trained using Gradient Descent. Seeing how gradient descent works on a simple example will build intuition and help us understand some of the nuances around setting the learning rate and other parameters. lets also explore Stochastic Gradient Descent and compare its behavior to the standard approach.

The next several cells have code to perform (full-batch) gradient descent.

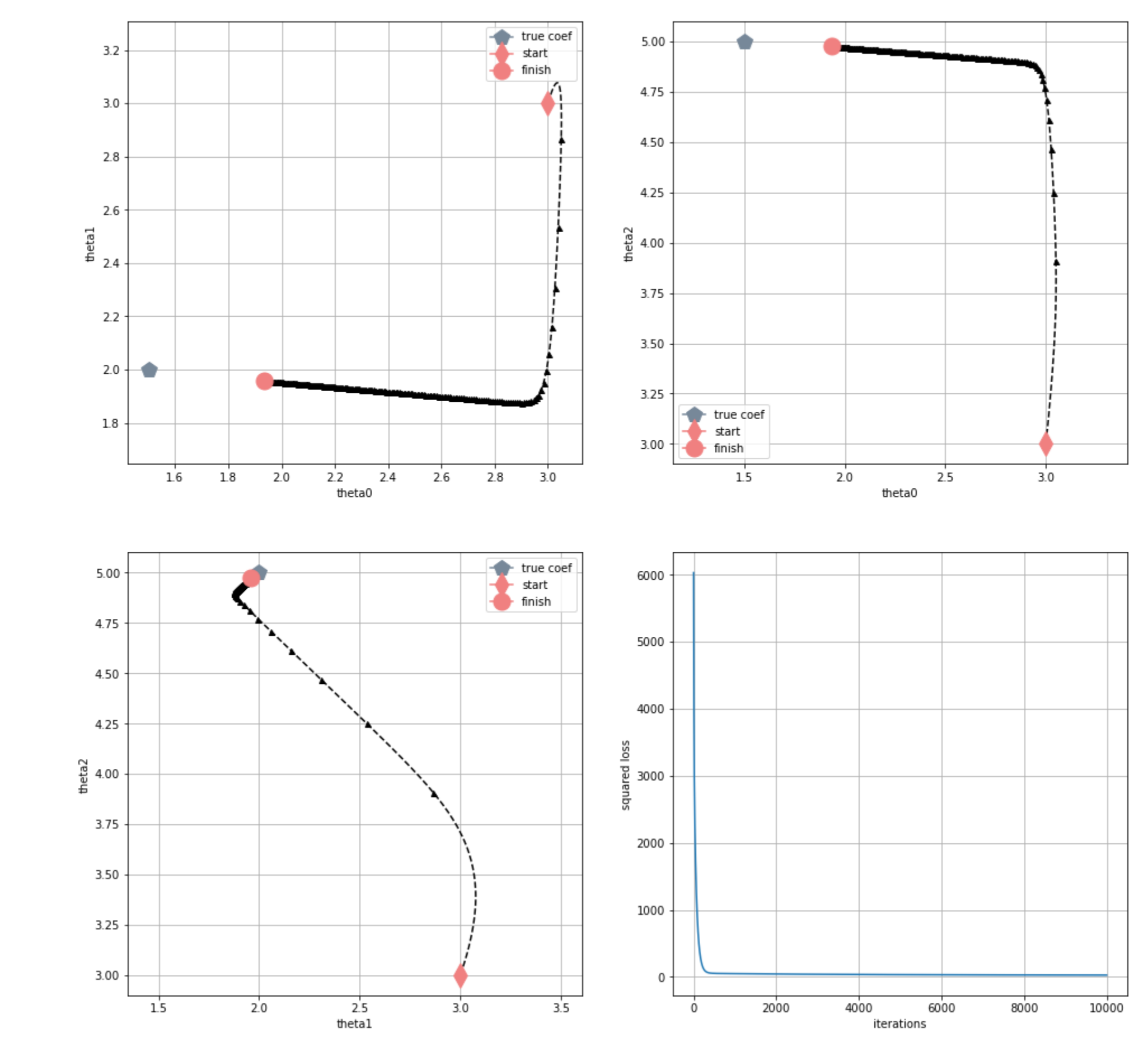
1. Picking a learning rate, and a number of iterations, run the code, and then plot the trajectory of your gradient descent.
2. Find examples where the learning rate is too high, too low, and "just right".
3. Look at plots of loss function under these conditions.

In [13]:
```python
## Parameters to play with
learning_rate = 1e-3
num_iter = 10000
theta_initial = np.array([3,3,3])
```

In [16]:
```python
def gradient_descent(learning_rate, num_iter, theta_initial):

    ## Initialization steps
    theta = theta_initial
    theta_path = np.zeros((num_iter+1,3))
    theta_path[0,:]= theta_initial

    loss_vec = np.zeros(num_iter)

    ## Main Gradient Descent loop (for a fixed number of iterations)
    for i in range(num_iter):
        y_pred = np.dot(theta.T,x_mat.T)
        loss_vec[i] = np.sum((y-y_pred)**2)
        grad_vec = (y-y_pred).dot(x_mat)/num_obs  #sum up the gradients across all observations and divide by num_obs
        grad_vec = grad_vec
        theta = theta + learning_rate*grad_vec
        theta_path[i+1,:]=theta
    return theta_path, loss_vec
```

In [20]:
```python
true_coef = [b, theta_1, theta_2]

def plot_ij(theta_path, i, j, ax):
    ax.plot(true_coef[i], true_coef[j],
            marker='p', markersize=15, label='true coef',
            color='#778899')
    ax.plot(theta_path[:, i],theta_path[:, j],
            color='k', linestyle='--', marker='^',
            markersize=5, markevery=50)
    ax.plot(theta_path[0, i], theta_path[0, j], marker='d',
            markersize=15, label='start', color='#F08080')
    ax.plot(theta_path[-1, i], theta_path[-1, j], marker='o',
            markersize=15, label='finish', color='#F08080')
    ax.set(
        xlabel='theta'+str(i),
        ylabel='theta'+str(j))
    ax.axis('equal')
    ax.grid(True)
    ax.legend(loc='best')


def plot_all(theta_path, loss_vec, learning_rate, num_iter, theta_initial, gdtype='Gradient Descent'):
    fig = plt.figure(figsize=(16, 16))
    title = '{gdtype} in the 3d parameter space - Learning rate is {lr} // {iters} iters // starting point {initial}'
    title = title.format(gdtype=gdtype, lr=learning_rate,
                         iters=num_iter, initial=theta_initial)
    fig.suptitle(title, fontsize=20)
    ax = fig.add_subplot(2, 2, 1)
    plot_ij(theta_path, 0, 1, ax)
    ax = fig.add_subplot(2, 2, 2)
    plot_ij(theta_path, 0, 2, ax)
    ax = fig.add_subplot(2, 2, 3)
    plot_ij(theta_path, 1, 2, ax)
    ax = fig.add_subplot(2, 2, 4)
    ax.plot(loss_vec)
    ax.set(xlabel='iterations', ylabel='squared loss')
    ax.grid(True)


theta_path, loss_vec = gradient_descent(learning_rate, num_iter, theta_initial)
plot_all(theta_path, loss_vec, learning_rate, num_iter, theta_initial)
```

Gradient Descent in the 3d parameter space - Learning rate is 0.001 // 10000 iters // starting point [3 3 3]



Stochastic Gradient Descent

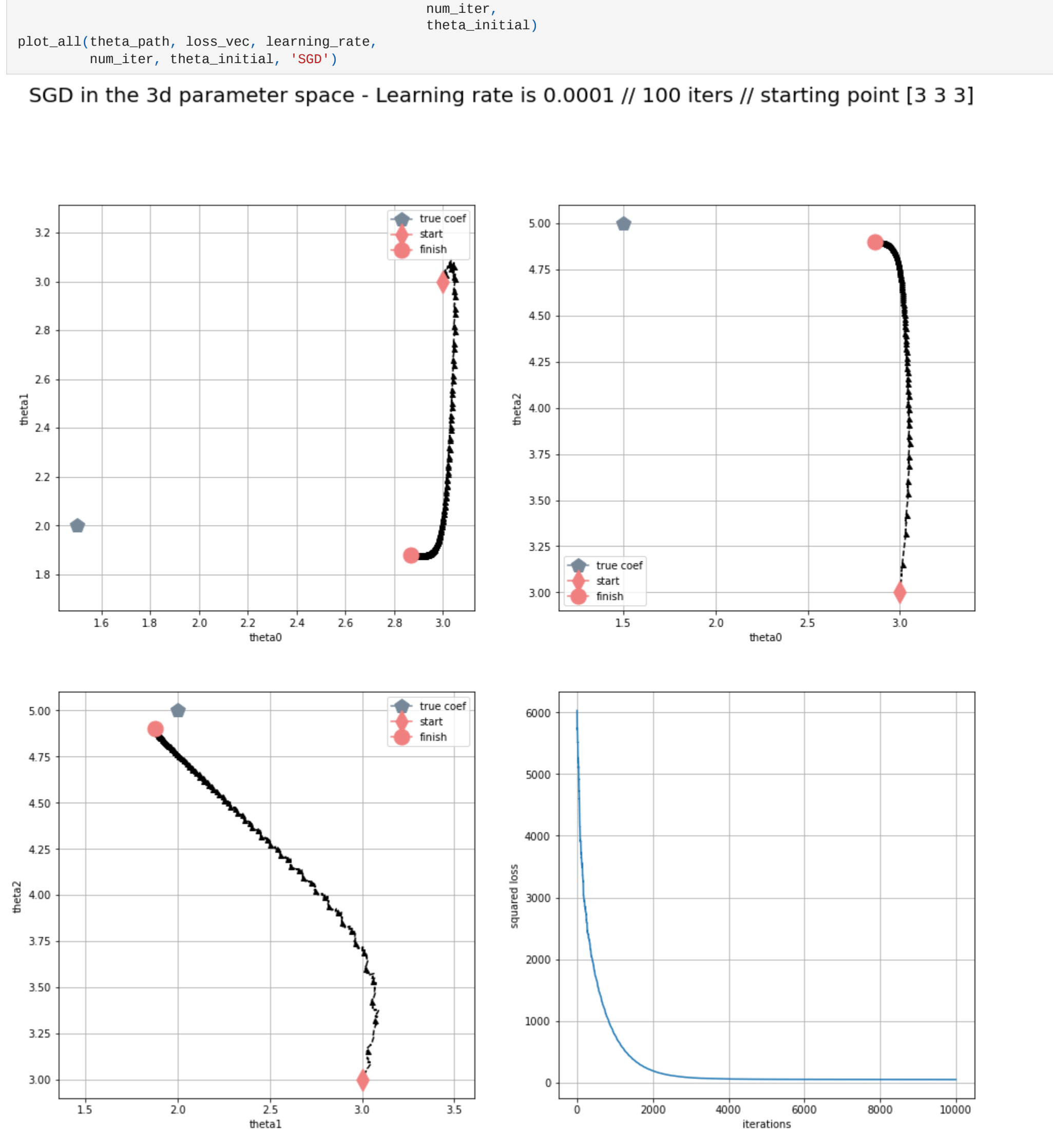
Rather than average the gradients across the whole dataset before taking a step, we will now take a step for every datapoint. Each step will be somewhat of an "overreaction" but they should average out.

The below code runs Stochastic Gradient descent, but runs through the data in the same order every time.

1. Running the code and plotting the graphs.
2. Modifying the code so that it randomly re-orders the data.

In [23]:
```python
def stochastic_gradient_descent(learning_rate, num_iter, theta_initial):

    ## Initialization steps
    theta = theta_initial
    # below are different in STOCHASTIC gradient descent
    theta_path = np.zeros(((num_iter*num_obs)+1,3))
    theta_path[0,:] = theta_initial
    loss_vec = np.zeros(num_iter*num_obs)

    ## Main SGD loop
    count = 0
    for i in range(num_iter):
        for j in range(num_obs):
            count+=1
            y_pred = np.dot(theta.T,x_mat.T)
            loss_vec[count-1] = np.sum((y-y_pred)**2)
            grad_vec = (y[j]-y_pred[j])*(x_mat[j,:])
            theta = theta + learning_rate*grad_vec
            theta_path[count,:]=theta
    return theta_path, loss_vec

## Parameters to play with
learning_rate = 1e-4
num_iter = 100
theta_initial = np.array([3, 3, 3])


theta_path, loss_vec = stochastic_gradient_descent(learning_rate,
                                                   num_iter,
                                                   theta_initial)
plot_all(theta_path, loss_vec, learning_rate,
         num_iter, theta_initial, 'SGD')
```

SGD in the 3d parameter space - Learning rate is 0.0001 // 100 iters // starting point [3 3 3]



lets play with the parameters below and observe the trajectory it results in

In [26]:
```python
## Parameters to play with
learning_rate = 1e-4
num_iter = 100
theta_initial = np.array([3,3,3])


theta_path, loss_vec = stochastic_gradient_descent(learning_rate,
                                                   num_iter,
                                                   theta_initial)
plot_all(theta_path, loss_vec, learning_rate, num_iter,
         theta_initial, 'SGD')
```

SGD in the 3d parameter space - Learning rate is 0.0001 // 100 iters // starting point [3 3 3]