6.390 Introduction to Machine Learning

Recitation Week #3

Issued September 18, 2023

1. Joe Smallbucks want to build a stock portfolio with an estimated budget of \$1000. Joe is considering a set of n stocks with known prices. His task is to decide how much of each stock, $x = \begin{bmatrix} x_1, & x_2, & \ldots, & x_n \end{bmatrix}^\top$, to hold. Let p_i be the price of one unit of stock i, and $p = \begin{bmatrix} p_1, & p_2, & \ldots, & p_n \end{bmatrix}^\top$. If Joe holds x_i units of stock i, we say that the value of that stock in Joe's portfolio is $x_i p_i$. The value of Joe's whole portfolio is the sum of the values of all individual stocks. Note that x_i can be positive or negative (also known as "long" or "short" positions, respectively).

   Joe hopes that he can find an optimal portfolio by using gradient descent.

   (a) Joe first seeks to formulate his objective, $J(x)$. He decides to impose a "budget penalty" for being over or under his budget that is 0.1 times the square of how much his total portfolio value differs from his budget (\$1000). Joe also defines a "share holding penalty" that squares the x_i for each stock in his portfolio, then takes the total sum of these, and weights this sum by 0.5. His final objective is the sum of his budget penalty and his share holding penalty.

   Write an expression for the objective function that Joe is seeking to minimize, in terms of the **vectors** x and p.

   (b) To perform gradient descent, Joe needs the gradient of J with respect to his decision variable, x. Derive an expression for $\nabla_x J(x)$ in terms of x and p.

   (c) Let's try out gradient descent, with a step size of $\eta = 0.1$. We consider a particular case where $p = [1, 3]^\top$, and our initial guess for holdings is $x = [0, 0]^\top$.

   After one iteration of gradient descent, what is x? Show your work.

2. In this problem, we will study the effects of different hyperparameter choices on the behavior of gradient descent in the context of linear regression. For simplicity, we will ignore the offset, assume $\theta_0 = 0$. Our hypothesis has the form $h(x;\theta) = \theta^\top x$. Our objective function has the form:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(h(x^{(i)};\theta) - y^{(i)}\right)^2 + \lambda\|\theta\|^2.$$

Recall that one iteration of gradient descent is defined as:

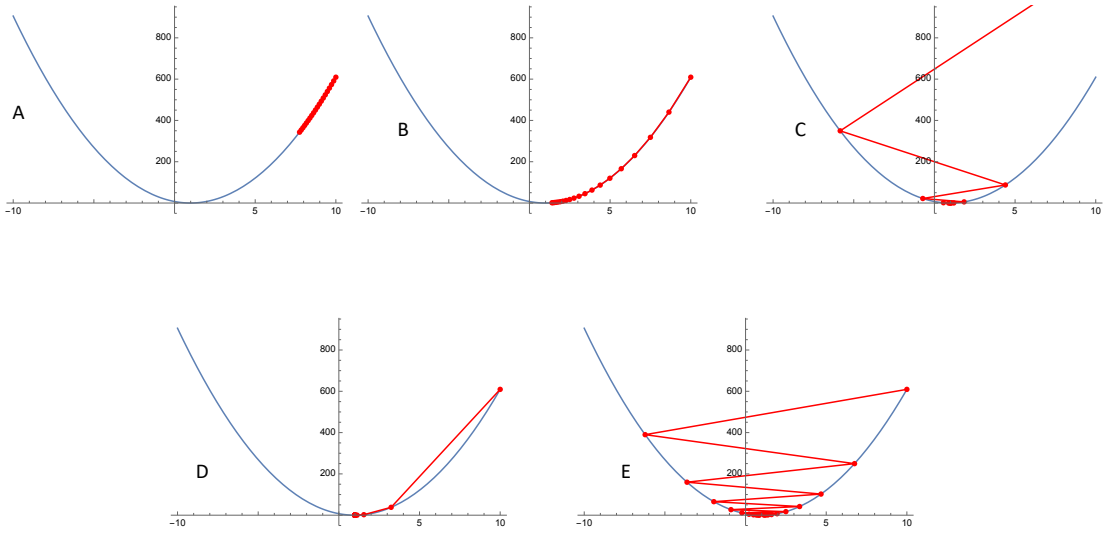
$$\theta^{(t+1)} = \theta^{(t)} - \eta\,\nabla_\theta J(\theta)|_{\theta=\theta^{(t)}}\,,$$

where η is the step size.

(a) Consider this 1D dataset with 4 data points:

```
X = np.array([[1, 2, 3, 4]])
y = np.array([1.2, 1.8, 3.2, 3.8])
```

Here are plots of the objective function for $\lambda = 0$ with various trajectories of gradient descent overlaid. The trajectories may vary by the choice of the step size and starting guess.



i. Would you expect to be able to find the global minimum using gradient descent? Why or why not?

ii. For each of the plots on the previous page, mark all of the following descriptions that apply:

1. η is 0.001, starting guess is 10
2. η is 0.01, starting guess is 10
3. η is 0.05, starting guess is 10
4. η is 0.2, starting guess is 2
5. η is 0.12, starting guess is 10
6. converges to minimum, no oscillation
7. converges to minimum, with oscillation
8. does not reach the minimum
9. diverges

(b) We will refer to the following function calls for learning a linear regressor:

```
def lin_reg_analytic(X,y,lambda):
    ... # Implemented as presented in Lecture Notes Section 2.5
    return (theta, MSE)

def regress_gd(X, y, lambda, step_size, num_steps, init):
    ... # Implemented as presented in Lecture Notes Section 3.3
    return (theta, MSE)
```

Consider the following 2D dataset:

```
X = np.array([[1, 2, 3, 4],
              [2, 1, 4, 3]])
y = np.array([1.2, 1.8, 3.2, 3.8])
```

We first compute the analytic solution without regularization and then run three instances of gradient descent with different values for the step size:

```
>>> lin_reg_analytic(X,y,0)
(array([0.8, 0.2]), 0)

>>> regress_gd(X, y, 0.00, 0.1, 1000, np.array([ 0.0, 0.0]))
(array([-2.83511681e+278, -2.83511681e+278]), inf)
>>> regress_gd(X, y, 0.00, 0.01, 1000, np.array([ 0.0, 0.0]))
(array([0.79998705, 0.20001295]), 1.67738094269143e-10)
>>> regress_gd(X, y, 0.00, 0.001, 1000, np.array([ 0.0, 0.0]))
(array([0.68969137, 0.31030863]), 0.012167993285775044)
```

Compare the output between the three executions of gradient descent with different hyperparameters; which experiment(s) converged? Which diverged, if any?

(c) Here are two slightly different datasets containing four data points each (note that data points are columns) and two vectors containing the corresponding labels:

```
X1 = np.array([[1, 2, 3, 4],
               [1, 2, 3, 4]])
X2 = np.array([[1, 2, 3, 4],
               [1.000001, 2.000002, 3.000003, 4.00001]])
y1 = np.array([1, 2, 3, 4])
y2 = np.array([1.2, 1.8, 3.2, 3.8])
```

We run the following function calls to compute the analytic solution without regularization:

```
>>> lin_reg_analytic(X2,y1,0)
(array([ 9.99961689e-01, -3.63133399e-05]), 1.6706405582993305e-07)
>>> lin_reg_analytic(X2,y2,0)
(array([ 52385.11668617, -52384.03587999]), 0.051556090396712245)
```

i. What would happen if we tried to execute: `lin_reg_analytic(X1,y1,0)`?

ii. Compare the results of running `lin_reg_analytic(X2,y1,0)` and `lin_reg_analytic(X2,y2,0)`. What behavior are we observing when attempting to compute the analytic solution?

iii. Consider the linear regressor hypothesis using the parameters learned by executing `lin_reg_analytic(X2,y2,0)`. What is the predicted label the data point `[1, 1]`? How about `[1.01, 1.02]`? Is that good?

(d) Let's consider the effect of regularization when running gradient descent, on the same datasets from part (c); the function definitions are repeated for your convenience:

```
def lin_reg_analytic(X,y,lambda):
    ... # Implemented as presented in Lecture Notes Section 2.5
    return (theta, MSE)

def regress_gd(X, y, lambda, step_size, num_steps, init):
    ... # Implemented as presented in Lecture Notes Section 3.3
    return (theta, MSE)

>>> lin_reg_analytic(X2, y2, 0.01)
array([0.49299887, 0.49301017])

>>> regress_gd(X2, y2, 0.01, 0.01, 100000, np.array([ 10, -10]))
# final gradient [ 4.11489818e-10 -4.11489180e-10]
(array([0.49299889, 0.49301015]), 0.04353086796908341)
>>> regress_gd(X2, y2, 0.0, 0.01, 100000, np.array([ 10, -10]))
# final gradient [ 1.10010018e-07 -1.10009880e-07]
(array([10.49322603, -9.50655365]), 0.03866875820336399)
```

The first experiment sets the hyperparameter λ to 0.01, while the second does not include regularization. What impact does regularization have on learning a hypothesis for this dataset?

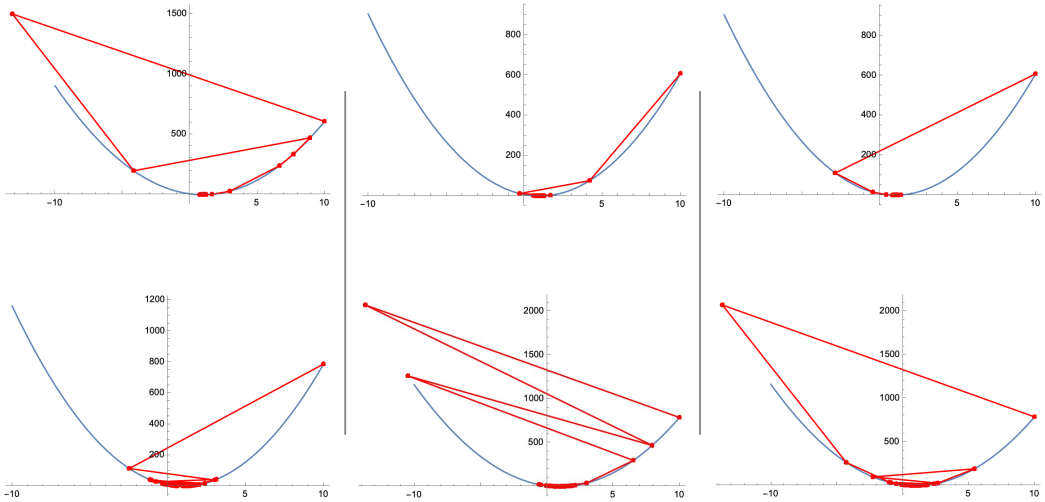
3. Consider solving an unregularized linear regression problem using *stochastic gradient descent*. For simplicity, we will ignore the offset, assume $\theta_0 = 0$. Our hypothesis has the form, $h(x;\theta) = \theta^\top x$. Our objective function has the form:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(h(x^{(i)};\theta) - y^{(i)}\right)^2.$$

Consider two simple 1D datasets:

```
X1 = np.array([[1, 2, 3, 4]])
y1 = np.array([1.2, 1.8, 3.2, 3.8])
```

```
X2 = np.array([[1, 2, 3, 4]])
y2 = np.array([2, 1, 4, 3])
```



Each row of plots corresponds to various executions of SGD on one of the respective data sets, with a fixed step size of 0.08 and a starting guess of 10.

(a) Why do these trajectories "jump around" so much?

(b) Why do the trajectories in the second row "jump around" more than those in the first row? (Hint: plot both datasets.)