CS168, Spring 2024

Mini-Project #3

Due by 11am Thursday, April 25th.

Instructions

- If you work in a group (of at most four students), please submit one assignment via Gradescope (please have all group members register on gradescope, and link all group members to your submission).
- Detailed submission instructions can be found on the course website (`https://web.stanford.edu/class/cs168`) under "Coursework - Assignments" section.
- Use 12pt or higher font for your writeup.
- Make sure the plots you submit are easy to read at a normal zoom level.
- If you've written code to solve a certain part of a problem, or if the part explicitly asks you to implement an algorithm, you must also include the code in your pdf submission.
- Code marked as Deliverable should be pasted into the relevant section. Keep variable names consistent with those used in the problem statement, and with general conventions. No need to include import statements and other scaffolding, if it is clear from context. Use the `verbatim` or "minted" environment to paste code in LaTeX.

```
def example():
     print "Your code should be formatted like this."
```

- **Reminder:** No late assignments will be accepted, but we will drop your lowest assignment grade.

Goal of mini-project: In the three problems of this mini-project, you will explore the idea of *generalization*, i.e., when the test error of a learned prediction function is roughly the same as its training error. You will explore how regularization and the choice of the learning algorithm (gradient descent, stochastic gradient descent, etc.) interact with generalization in a simple linear prediction setting.[1] In the last problem, you will see a surprising example of "double-descent"—the recently highlighted phenomenon where more data might actually lead to worse performance. Many aspects of these relationships are still not well understood, and a fierce debate is currently raging within the Machine Learning community about whether our understanding of generalization lacks key components. This week will give you a glimpse of some of these mysteries.

Part 1: Regression, Three Ways

We will consider the problem of fitting a linear model. Given d-dimensional input data $\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(n)} \in \mathbb{R}^d$ with real-valued labels $y^{(1)}, \cdots, y^{(n)} \in \mathbb{R}$, the goal is to find the coefficient vector $\mathbf{a}$ that minimizes the sum of the squared errors. The total squared error of $\mathbf{a}$ can be written as $f(\mathbf{a}) = \sum_{i=1}^{n} f_i(\mathbf{a})$, where $f_i(\mathbf{a}) = (\mathbf{a}^\top \mathbf{x}^{(i)} - y^{(i)})^2$ denotes the squared error of the ith data point.

The data in this problem will be drawn from the following linear model. For the training data, we select n data points $\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(n)}$, each drawn independently from a d-dimensional Gaussian distribution. We then

[1] Our understanding is that many of you have already seen gradient descent and stochastic gradient descent in multiple other classes. If you haven't, and with to review these algorithms see, for example, the notes for Lectures #5 and #6 of the 2016 offering of CS168, available at `https://web.stanford.edu/class/cs168/l/gradDescNotes.pdf` and `https://web.stanford.edu/class/cs168/l/stochDesc16.pdf`.

pick the "true" coefficient vector $\mathbf{a}^*$ (again from a d-dimensional Gaussian), and give each training point $\mathbf{x}^{(i)}$ a label equal to $(\mathbf{a}^*)^\top \mathbf{x}^{(i)}$ plus some noise (which is drawn from a 1-dimensional Gaussian distribution).[2]

The following Python code will generate the data used in this problem.

```
d = 100 # dimensions of data
n = 1000 # number of data points
X = np.random.normal(0,1, size=(n,d))
a_true = np.random.normal(0,1, size=(d,1))
y = X.dot(a_true) + np.random.normal(0,0.5,size=(n,1))
```

(a) (4 points) Least-squares regression has the closed form solution $\mathbf{a} = (\mathbf{X}^\top \mathbf{X})^{-\mathbf{1}} \mathbf{X}^\top \mathbf{y}$, which minimizes the squared error on the data. (Here $\mathbf{X}$ is the $n \times d$ data matrix as in the code above, with one row per data point, and $\mathbf{y}$ is the n-vector of their labels.) Solve for $\mathbf{a}$ and report the value of the objective function using this value $\mathbf{a}$. For comparison, what is the total squared error if you just set $\mathbf{a}$ to be the all 0's vector?

Comment: Computing the closed-form solution requires time $O(nd^2 + d^3)$, which is slow for large d. Although gradient descent methods will not yield an exact solution, they do give a close approximation in much less time. For the purpose of this assignment, you can use the closed form solution as a good sanity check in the following parts.

(b) (6 points) In this part, you will solve the same problem via gradient descent on the squared-error objective function $f(\mathbf{a}) = \sum_{i=1}^{n} f_i(\mathbf{a})$. Recall that the gradient of a sum of functions is the sum of their gradients. Given a point $\mathbf{a}_t$, what is the gradient of f at $\mathbf{a}_t$?

Now use gradient descent to find a coefficient vector $\mathbf{a}$ that approximately minimizes the least squares objective function over the data. Run gradient descent three times, once with each of the step sizes 0.00005, 0.0005, and 0.0007. You should initialize $\mathbf{a}$ to be the all-zero vector for all three runs. Plot the objective function value for 20 iterations for all 3 step sizes on the same graph. Comment in 3-4 sentences on how the step size can affect the convergence of gradient descent (feel free to experiment with other step sizes). Also report the step size that had the best final objective function value and the corresponding objective function value. *In this part, write your own gradient descent code—it will just two or three lines of code—do not use any package that does this for you!!*

(c) (6 points) In this part you will run *stochastic gradient descent* (SGD) to solve the same problem. Recall that in stochastic gradient descent, you pick one datapoint at a time, say $(\mathbf{x}^{(i)}, y^{(i)})$, and update your current value of $\mathbf{a}$ according to the gradient of $f_i(\mathbf{a}) = (\mathbf{a}^\top \mathbf{x}^{(\mathbf{i})} - y^{(i)})^2$.

Run stochastic gradient descent using step sizes $\{0.0005, 0.005, 0.01\}$ and 1000 iterations. Plot the objective function value vs. the iteration number for all 3 step sizes on the same graph. Comment 3-4 sentences on how the step size can affect the convergence of stochastic gradient descent and how it compares to gradient descent. Compare the performance of the two methods. How do the best final objective function values compare? How many times does each algorithm use each data point? Also report the step size that had the best final objective function value and the corresponding objective function value. *In this part, write your own SGD code—it will just be a few lines of code—do not use any package that does this for you!!*

Deliverables: Objective function value for part (a). Gradient calculation for part (b). Code, plot, discussion and optimal step size and objective function value for parts (b) and (c).

Part 2

In the previous problem, the number of data points was much larger than the number of dimensions and hence we did not worry about generalization. (Feel free to check that the coefficient vector $\mathbf{a}$ that you

[2] Test data will be drawn from the same distribution, but we won't worry about this until Part 2.

computed accurately labels new datapoints drawn from the same distribution.) We will now consider the setting where $d = n$, and examine the test error along with the training error. Use the following Python code for generating the training data and test data.

```
train_n = 100
test_n = 1000
d = 100
X_train = np.random.normal(0,1, size=(train_n,d))
a_true = np.random.normal(0,1, size=(d,1))
y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(train_n,1))
X_test = np.random.normal(0,1, size=(test_n,d))
y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(test_n,1))
```

(a) (2 points) We will first setup a baseline, by finding the test error of the linear regression solution $\mathbf{a} = \mathbf{X^{-1}y}$ without any regularization. This is the closed-form solution for the minimizer of the objective function $f(\mathbf{a})$. (Note the formula is simpler than in 1(a) because now $\mathbf{X}$ is square.) Report the training error and test error of this approach, averaged over 10 trials. For better interpretability, report the normalized test error $\hat{f}(\mathbf{a})$ rather than the value of the objective function $f(\mathbf{a})$, where by definition

$$\hat{f}(\mathbf{a}) = \frac{\| \mathbf{Xa} - \mathbf{y} \|_2}{\| \mathbf{y} \|_2}.$$

(b) (5 points) We will now examine ℓ_2 regularization as a means to prevent overfitting. The ℓ_2 regularized objective function is given by the following expression:

$$\sum_{i=1}^{m} (\mathbf{a}^\top \mathbf{x^{(i)}} - y^{(i)})^2 + \lambda \parallel \mathbf{a} \parallel_2^2 .$$

This has a closed-form solution $\mathbf{a} = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{\mathbf{-1}}\mathbf{X}^\top\mathbf{y}$. Using this closed-form solution, present a plot of the normalized training error and normalized test error $\hat{f}(\mathbf{a})$ for $\lambda = \{0.0005, 0.005, 0.05, 0.5, 5, 50, 500\}$. As before, you should average over 10 trials. Discuss the characteristics of your plot, and also compare it to your answer to (a).

(c) (5 points) Run stochastic gradient descent (SGD) on the original objective function $f(\mathbf{a})$, with the initial guess of $\mathbf{a}$ set to be the all 0's vector. Run SGD for 1,000,000 iterations for each different choice of the step size, $\{0.00005, 0.0005, 0.005\}$. Report the normalized training error and the normalized test error for each of these three settings, averaged over 10 repetitions/trials. How does the SGD solution compare with the solutions obtained using ℓ_2 regularization? Note that SGD is minimizing the original objective function, which does *not* have any regularization. In Part (a) of this problem, we found the *optimal* solution to the original objective function with respect to the training data. How does the training and test error of the SGD solutions compare with those of the solution in (a)? Can you explain your observations? (It may be helpful to also compute the normalized training and test error corresponding to the true coefficient vector $f(\mathbf{a}^*)$, for comparison.)

(d) (7 points) We will now examine the behavior of SGD in more detail. For step sizes $\{0.00005, 0.005\}$ and 1,000,000 iterations of SGD,

(i) Plot the normalized training error vs. the iteration number. On the plot of training error, draw a line parallel to the x-axis indicating the error $\hat{f}(\mathbf{a}^*)$ of the true model $\mathbf{a}^*$.

(ii) Plot the normalized test error vs. the iteration number. Your code might take a long time to run if you compute the test error after every SGD step—feel free to compute the test error every 100 iterations of SGD to make the plots.

(iii) Plot the ℓ_2 norm of the SGD solution vs. the iteration number.

Comment on the plots. What can you say about the generalization ability of SGD with different step sizes? Does the plot correspond to the intuition that a learning algorithm starts to overfit when the training error becomes too small, i.e. smaller than the noise level of the true model? How does the generalization ability of the final solution depend on the ℓ_2 norm of the final solution?

(e) (4 points) We will now examine the effect of the starting point on the SGD solution. Fixing the step size at 0.00005 and the maximum number of iterations at 1,000,000, choose the initial point randomly from the d-dimensional sphere with radius $r = \{0, 0.1, 0.5, 1, 10, 20, 30\}$, and plot the average normalized training error and the average normalized test error over 10 trials vs r. Comment on the results, in relation to the results from part (b) where you explored different ℓ_2 regularization coefficients. Can you provide an explanation for the behavior seen in this plot?

Deliverables: Code for all parts. Training and test error for part (a). Plots for part (b), (d) and (e). Training and test error for different step sizes for part (c). Explanation for parts (b), (c), (d), (e).

Part 3

In this part, you will explore "double-descent", a surprising phenomena whereby the performance of a model improves, then worsens, then improves, as the amount of training data increases. This has been intensely debated and discussed over the past five years, and is currently a hot topic in machine learning and deep learning. (See the OpenAI blog post https://openai.com/blog/deep-double-descent/ for a discussion of this in the context of deep neural networks.) [Note: Double-descent is often formulated in terms of the effect of increasing the "capacity" or expressivity of the model, though the perspective we present here is a bit cleaner and corresponds to the same phenomenon.]

The setting we will consider is almost identical to the one in the first two parts, but instead of a regression problem, we will consider a linear classification problem where the data labels are ±1 according to which side of a hyperplane they lie. Python code for generating the training data and test data is below. Make sure you understand what this code is doing.

```
train_n = X [this will be varied between runs]
test_n = 1000
d = 500
X_train = np.random.normal(0,1, size=(train_n,d))
a_true = np.random.normal(0,1, size=(d,1))
y_train = np.sign(X_train.dot(a_true))
X_test = np.random.normal(0,1, size=(test_n,d))
y_test = np.sign(X_test.dot(a_true))
```

Consider the following learning algorithm:

1. Solve the ℓ_2 regularized *regression* problem where you try to predict the label as a linear function of $\mathbf{x}$:

$$\mathbf{a} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-\mathbf{1}} \mathbf{X}^\top \mathbf{y},$$

2. Given $\mathbf{a}$, your prediction of the label for a datapoint $\mathbf{x}$ is $sign(\mathbf{a}^\top \mathbf{x})$.

(a) (3 points) Lets see how well the above algorithm performs as the number of training points, *train_n*, varies, and the regularization parameter varies:

- Set the regularization parameter, $\lambda = 0$. For each value of $train\_n = 10, 20, 30, \ldots, 2d - 10, 2d$, run the above algorithm and record the test error (the fraction of the $test_n$ points where your predicted label is wrong). Repeat this 10 times and plot the average (over the 10 runs) test error as a function of the training set size *train_n*. [This should take under a minute to run.]

- Repeat the above for each value of $\lambda = 0.000001, 0.0001, 0.01, 1, 100$, and $10,000$, and plot the resulting curves on the same plot as the $\lambda = 0$ curve. [Hint: You should see a clear "double-descent" plot for at least some of these values of λ.]

(b) (5 points) Are you surprised by any of the plots from the previous part? Discuss the main features of the plots, and try to explain why these features make sense: For the $\lambda = 0$ plot, in hindsight, should we have expected this? For the curves that exhibit "double-descent", what is a plausible explanation for why the regularization results in these curves? For the largest λ curves, might we have expected this? [There is no one answer that we are expecting here. And do feel free to support your explanations/hypotheses with an additional experiment or two.]

(c) (3 points) In light of your understanding of generalization and regularization, what sorts of learning algorithms/settings would you expect to give rise to "double-descent" (e.g. where performance isn't monotonically improving as the number of training points increases)? What sorts of settings/learning algorithms would likely not give rise to "double-descent"?

Deliverables: Code, plots for part (a) and discussion for parts (b) and (c).