NYU | Center for Data Science

DS-GA 1003: Machine Learning (Spring 2020)

Homework 2: Ridge Regression and Gradient Descent

Due: Saturday, February 22, 2020 at 12pm

In this problem set, you will implement ridge regression using gradient descent and stochastic gradient descent. Along the way, you will get some practice with the surrounding probability concepts.

Instructions. You should upload your questions along with plots and code to Gradescope. You must follow the policies for submission detailed in Homework 0. Please map the Gradescope entry in the rubric to your name and NetId.

Dataset. We have provided you with a file in the `shared/` directory called `ridge_regression_dataset.csv`. Columns `x0` through `x47` correspond to the input and column `y` corresponds to the output.

Please check the supporting code in `skeleton_code.py` accessible on JupyterHub or GitHub. Throughout the problem set, we refer to particular blocks of code to help you step by step.

Computing Risk

1 Let $\vec{x}$ denote the random vector in $\mathbb{R}^n$ where each coordinate is i.i.d., taking the values $-2, 1, 0, +1, +2$ with equal probability (1/5 each).

(a) Compute $\mathbb{E}[||\vec{x}||_2^2]$.

(b) Compute $\mathbb{E}[||\vec{x}||_\infty]$. (where $||\vec{x}||_\infty = \max_{\forall i} |x_i|$)

(c) Compute the covariance matrix of $\vec{x}$.

2 (a) Let y be a random variable with a known distribution, and consider the square loss function $\ell(a, y) = (a - y)^2$. We want to find the action a^* that has minimal risk, namely, to find $a^* = \arg\min_a \mathbb{E}\,(a - y)^2$, where the expectation is with respect to y. Show that $a^* = \mathbb{E}y$, and the Bayes risk (i.e. the risk of a^*) is $\mathrm{Var}(y)$.

In other words, if you want to try to predict the value of a random variable, the best you can do (for minimizing expected square loss) is to predict the mean of the distribution. Your expected loss for predicting the mean will be the variance of the distribution. You should use the fact that $\mathrm{Var}(y) = \mathbb{E}y^2 - (\mathbb{E}y)^2$.

(b) Now let's introduce an input. Recall that the **statistical risk** of a decision function $f : \mathcal{X} \to \mathcal{A}$ is

$$R(f) = \mathbb{E}\ell(f(x), y),$$

where $(x, y) \sim P_{\mathcal{X}\times\mathcal{Y}}$, and the **target function** (sometimes called the Bayes decision function) $f^* : \mathcal{X} \to \mathcal{A}$ is a function that achieves the *minimal risk* among all possible functions:

$$R(f^*) = \inf_f R(f).$$

Here we consider the regression setting, in which $\mathcal{A} = \mathcal{Y} = \mathbf{R}$. We will show for the square loss $\ell(a, y) = (a - y)^2$, the Bayes decision function is $f^*(x) = \mathbb{E}\,[y \mid x]$, where the expectation is over y. As before, we assume we know the data-generating distribution $P_{\mathcal{X}\times\mathcal{Y}}$.

i. We'll approach this problem by finding the optimal action for any given x. If somebody tells us x, we know that the corresponding y is coming from the conditional distribution $y \mid x$. For a particular x, what value should we predict (i.e. what action a should we produce) that has minimal expected loss? Express your answer as a decision function $f(x)$, which gives the best action for any given x. In mathematical notation, we're looking for $f^*(x) = \arg\min_a \mathbb{E}\left[(a-y)^2 \mid x\right]$, where the expectation is with respect to y.

ii. In the previous problem we produced a decision function $f^*(x)$ that minimized the risk for each x. In other words, for any other decision function $f(x)$, $f^*(x)$ is going to be at least as good as $f(x)$, for every single x. So

$$\mathbb{E}\left[(f^*(x)-y)^2 \mid x\right] \le \mathbb{E}\left[(f(x)-y)^2 \mid x\right],$$

for all x. To show that $f^*(x)$ is the Bayes decision function, we need to show that

$$\mathbb{E}\left[(f^*(x)-y)^2\right] \le \mathbb{E}\left[(f(x)-y)^2\right]$$

for any f. Use the law of iterated expectations to show why this is true.

Linear Regression

1 When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values are treated as "more important", which is not usually what you want.

One common approach to feature normalization is perform an affine transformation (i.e. shift and rescale) on each feature so that all feature values in the training set are in $[0,1]$. Each feature gets its own transformation. We then apply the same transformations to each feature on the validation set or test set. It's important that the transformation is "learned" on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the $[0,1]$ interval.

(a) Modify function `feature_normalization` to normalize all the features to $[0,1]$. Can you use numpy's broadcasting here? Often broadcasting can help to simplify and/or speed up your code. Note that a feature with constant value cannot be normalized in this way. Your function should discard features that are constant in the training set.

2 In linear regression, we consider the hypothesis space of linear functions $h_\theta : \mathbf{R}^d \to \mathbf{R}$, where

$$h_\theta(x) = \theta^T x,$$

for $\theta, x \in \mathbf{R}^d$, and we choose θ that minimizes the following "average square loss" objective function:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x_i) - y_i\right)^2,$$

where $(x_1, y_1), \ldots, (x_m, y_m) \in \mathbf{R}^d \times \mathbf{R}$ is our training data.

While this formulation of linear regression is very convenient, it's more standard to use a hypothesis space of "affine" functions:

$$h_{\theta,b}(x) = \theta^T x + b,$$

which allows a nonzero intercept term – sometimes called a "bias" term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to x that is always a fixed value, such as 1. You should convince yourself that this is equivalent. We'll assume this representation, and thus we'll actually take $\theta, x \in \mathbf{R}^{d+1}$.

(a) Let $X \in \mathbf{R}^{m\times(d+1)}$ be the **design matrix**, where the i'th row of X is x_i. Let $y = (y_1, \ldots, y_m)^T \in \mathbf{R}^{m\times 1}$ be the "response". Write the objective function $J(\theta)$ as a matrix/vector expression, without using an explicit summation sign. [Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.]

(b) Write down an expression for the gradient of J (again, as a matrix/vector expression, without using an explicit summation sign).

(c) In our search for a θ that minimizes J, suppose we take a step from θ to $\theta + \eta h$, where $h \in \mathbf{R}^{d+1}$ is the "step direction" (recall, this is not necessarily a unit vector) and $\eta \in (0, \infty)$ is the "step size" (note that this is not the actual length of the step, which is $\eta\|h\|$). Use the gradient to write down an approximate expression for the change in objective function value $J(\theta + \eta h) - J(\theta)$. [This approximation is called a "linear" or "first-order" approximation.]

(d) Write down the expression for updating θ in the gradient descent algorithm. Let η be the step size.

(e) Modify the function `compute_square_loss`, to compute $J(\theta)$ for a given θ. You might want to create a small dataset for which you can compute $J(\theta)$ by hand, and verify that your `compute_square_loss` function returns the correct value.

(f) Modify the function `compute_square_loss_gradient`, to compute $\nabla_\theta J(\theta)$. You may again want to use a small dataset to verify that your `compute_square_loss_gradient` function returns the correct value.

3 Recall from Lab 3 that we can numerically check the gradient calculation. If $J : \mathbf{R}^d \to \mathbf{R}$ is differentiable, then for any vector $h \in \mathbf{R}^d$, the directional derivative of J at θ in the direction h is given by

$$\lim_{\varepsilon\to 0} \frac{J(\theta + \varepsilon h) - J(\theta - \varepsilon h)}{2\epsilon}.$$

It is also given by the more standard definition of directional derivative,

$$\lim_{\varepsilon\to 0} \frac{1}{\varepsilon}\left[J(\theta + \varepsilon h) - J(\theta)\right] .$$

The form given gives a better approximation to the derivative when we are using small (but not infinitesimally small) ε. We can approximate this directional derivative by choosing a small value of $\varepsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. In other words, take $h = (1, 0, 0, \ldots, 0)$ to get the first component of the gradient. Then take $h = (0, 1, 0, \ldots, 0)$ to get the second component. And so on. See here for details.

(a) Complete the function `grad_checker` according to the documentation given. Alternatively, you may complete the function `generic_grad_checker so` that it works for any objective function. It should take as parameters a function that computes the objective function and a function that computes the gradient of the objective function. Note: Running the gradient checker takes extra time. In practice, once you're convinced your gradient calculator is correct, you should stop calling the checker so things run faster.

4 At the end of the skeleton code, the data is loaded, split into a training and test set, and normalized. We will now finish the job of running regression on the training set.

(a) Complete `batch_gradient_descent`. Note the phrase "batch gradient descent" distinguishes between gradient and stochastic gradient descent or more generally minibatch gradient descent.

(b) Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step sizes 0.5, 0.1, .05, and .01. Plot the average square loss as a function of the number of steps for each step size. Briefly summarize your findings.

(c) **(Optional)** Implement backtracking line search. How does it compare to the best fixed step-size you found in terms of number of steps? In terms of time? How does the extra time to run backtracking line search at each step compare to the time it takes to compute the gradient? (You can also compare the operation counts.)

Ridge Regression

We will add ℓ_2 regularization to linear regression. When we have a large number of features compared to instances, regularization can help control overfitting. Ridge regression is linear regression with ℓ_2 regularization. The regularization term is sometimes called a penalty term. The objective function for ridge regression is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2 + \lambda \theta^T \theta,$$

where λ is the regularization parameter, which controls the degree of regularization. Note that the offset term is being regularized as well. We will address that below.

1. Compute the gradient of $J(\theta)$ and write down the expression for updating θ in the gradient descent algorithm. (Matrix/vector expression – no summations please.)

2. Implement `compute_regularized_square_loss_gradient`.

3. Implement `regularized_grad_descent.`

4. For regression problems, we may prefer to leave the bias term unregularized. One approach is to change $J(\theta)$ so that the bias is separated out from the other parameters and left unregularized. Another approach that can achieve approximately the same thing is to use a very large number B, rather than 1, for the extra bias dimension. Explain why making B large decreases the effective regularization on the bias term, and how we can make that regularization as weak as we like (though not zero).

5. Now fix $B = 1$. Choosing a reasonable step-size (or using backtracking line search), find the θ_λ^* that minimizes $J(\theta)$ over a range of λ. You should plot the training average square loss and the test average square loss (just the average square loss part, without the regularization, in each case) as a function of λ. Your goal is to find λ that gives the minimum average square loss on the test set. It's hard to predict what λ that will be, so you should start your search very broadly, looking over several orders of magnitude. For example, $\lambda \in \left\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\right\}$. Once you find a range that works better, keep zooming in. You may want to have $\log(\lambda)$ on the x-axis rather than λ. [If you like, you may use sklearn to help with the hyperparameter search.]

6. What θ would you select in practice and why?

Stochastic Gradient Descent

When the training data set is very large, evaluating the gradient of the objective function can take a long time, since it requires looking at each training example to take a single gradient step. For **empirical risk**

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$$

stochastic gradient descent (SGD) can be very effective. In SGD, rather than taking $-\nabla J(\theta)$ as our step direction, we take $-\nabla f_i(\theta)$ for some i chosen uniformly at random from $\{1, \ldots, m\}$. The approximation is poor, but we will show it is unbiased.

In machine learning applications, each $f_i(\theta)$ would be the loss on the ith example (and of course we'd typically write n instead of m, for the number of training points). In practical implementations for ML, the data points are **randomly shuffled**, and then we sweep through the whole training set one by one, and perform an update for each training example individually. Recall from Homework 1 that one pass through the data is called an **epoch**. Note that each epoch of SGD touches as much data as a single step of batch gradient descent. You can use the same ordering for each epoch, though optionally you could investigate whether reshuffling after each epoch affects the convergence speed.

1. Show that the objective function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \left(h_\theta(x_i) - y_i\right)^2 + \lambda\theta^T\theta$$

can be written in the form $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$ by giving an expression for $f_i(\theta)$ that makes the two expressions equivalent.

2. Show that the stochastic gradient $\nabla f_i(\theta)$, for i chosen uniformly at random from $\{1,\ldots,m\}$, is an **unbiased estimator** of $\nabla J(\theta)$. In other words, show that $\mathbb{E}\left[\nabla f_i(\theta)\right] = \nabla J(\theta)$ for any θ. It will be easier to prove this for a general $J(\theta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\theta)$, rather than the specific case of ridge regression. You can start by writing down an expression for $\mathbb{E}\left[\nabla f_i(\theta)\right]$

3. Write down the update rule for θ in SGD for the ridge regression objective function.

4. Implement `stochastic_grad_descent`.

5. Use SGD to find θ^*_λ that minimizes the ridge regression objective for the λ and B that you selected in the previous problem. (If you could not solve the previous problem, choose $\lambda = 10^{-2}$ and $B = 1$). Try a few fixed step sizes (at least try $\eta_t \in \{0.05, .005\}$. Note that SGD may not converge with fixed step size. Simply note your results. Next try step sizes that decrease with the step number according to the following schedules: $\eta_t = \frac{C}{t}$ and $\eta_t = \frac{C}{\sqrt{t}}$, $C \leq 1$. Please include $C = 0.1$ in your submissions. You're encouraged to try different values of C (see notes below for details). For each step size rule, plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch (or step number, if you prefer) for each of the approaches to step size. How do the results compare?

   - In this case we are investigating the convergence rate of the optimization algorithm with different step size schedules, thus we're interested in the value of the objective function, which includes the regularization term.
   - Sometimes the initial step size (C for C/t and $C/\sqrt{t}$) is too aggressive and will get you into a part of parameter space from which you can't recover. Try reducing C to counter this problem.
   - As we'll learn in an upcoming lecture, SGD convergence is much slower than GD once we get close to the minimizer. (Remember, the SGD step directions are very noisy versions of the GD step direction). If you look at the objective function values on a logarithmic scale, it may look like SGD will never find objective values that are as low as GD gets. In terminology from Lecture 3, GD has much smaller "optimization error" than SGD. However, this difference in optimization error is usually dominated by other sources of error (estimation error and approximation error). Moreover, for very large datasets, SGD (or minibatch GD) is much faster (by wall-clock time) than GD to reach a point that's close [enough] to the minimizer.

6. **(Optional)** There are several variants of SGD.

   (a) Rather than using the last parameter value, say θ^T, we can use the average of all parameter values we visit along the optimization path: $\theta = \frac{1}{T}\sum_{t=1}^{T}\theta^t$, where T is total number of steps taken. Try this approach and see how it compares.

   (b) Try a stepsize rule of the form $\eta_t = \frac{\eta_0}{1+\eta_0\lambda t}$, where λ is your regularization constant, and η_0 a constant you can choose. How do the results compare?

Acknowledgement: This problem set is based on assignments developed by David Rosenberg of NYU and Bloomberg.