Problem Set 2: Gradient Descent & Regularization

Due: Tuesday, February 24, 2026 at 11:59pm ET

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX or MathJax in iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. For the coding problems, the text **Submission:** indicates all you need to submit in your PDF submission.

Problem 1: Descent Lemma (15 Points)

In this problem, we will derive the descent lemma. Recall from Lecture 2 the formal statement of the descent lemma: *if $F : \mathbb{R}^d \to \mathbb{R}$ is continuously twice-differentiable and L-smooth for any $w \in \mathbb{R}^d$, then for $0 < \eta \leq 1/L$,*

$$F(w - \eta \nabla F(w)) \leq F(w) - \frac{\eta}{2} \|\nabla F(w)\|^2.$$

Recall that an *L-smooth* function is one whose derivatives do not change too much:

$$\|\nabla F(x) - \nabla F(y)\| \leq \|x - y\| \quad \forall x, y \in \mathbb{R}^d.$$

Of course, the second derivative (the Hessian, for multivariate scalar-valued functions), is just the change in the first derivative. So L-smoothness should relate to a function's second derivatives.

For twice differentiable functions, L-smoothness is equivalent to a bound on the eigenvalues of its Hessian. If $\lambda_{\max}(\cdot)$ is the operator that takes the maximum eigenvalue of a matrix, then L-smoothness is just equivalent to

$$\lambda_{\max}(\nabla^2 F(x)) \leq L \quad \forall x \in \mathbb{R}^d,$$

where $\nabla^2 F(x) \in \mathbb{R}^{d \times d}$ is the Hessian of F and $L > 0$. We won't prove this equivalence, but you can take it on faith.

In words, the descent lemma says that, as long as F is "doesn't change too wildly," (smooth) gradient descent is guaranteed to decrease the objective value by at least $\frac{\eta}{2}\|\nabla F(w)\|^2$.

Recall from the "rough derivation" presented in lecture that we first used the definition of the derivative to get a first-order approximation. For a function $F : \mathbb{R}^d \to \mathbb{R}$ we saw that, for any point $w \in \mathbb{R}^d$,

$$F(v) \approx F(w) + \nabla F(w)^\top (v - w),$$

as long as v is close to w. If we pick a direction $\delta \in \mathbb{R}^d$ to move from w and consider what happens at $v = w + \delta$, we get

$$F(w + \delta) \approx F(w) + \nabla F(w)^\top \delta,$$

if δ is small (because then v would be close to w).

This wasn't a formal statement, but *Taylor's Theorem* from multivariable calculus makes this formal. It states that, for any $w, \delta \in \mathbb{R}^d$, there exists a $\tilde{w} \in \mathbb{R}^d$ on the line segment between w and $w + \delta$ such that

$$F(w + \delta) = F(w) + \nabla F(w)^\top \delta + \frac{1}{2}\delta^\top \nabla^2 F(\tilde{w})\delta.$$

Notice that the $\approx$ symbol is now replaced with how much the approximation is actually off by:

$$\frac{1}{2}\delta^\top \nabla^2 F(\tilde{w})\delta.$$

Functions of the form

$$g(v) = v^\top A v,$$

are called *quadratic forms* and they are the multivariate equivalent of the leading quadratic term in a quadratic function (i.e. ax^2 in $ax^2 + bx + c$). A very useful fact about quadratic forms is that

$$\max_{v \in \mathbb{R}^d : \|v\| = 1} v^\top A v = \lambda_{\max}(A).$$

In words, maximizing $v^\top A v$ over the unit vector directions gives the largest eigenvalue of A.

Problem 1(a) (5 points)

Suppose that F is L-smooth. Using the above fact about quadratic forms and Taylor's Theorem, prove that, for any $w, \delta \in \mathbb{R}^d$, we have

$$F(w + \delta) \leq F(w) + \nabla F(w)^\top \delta + \frac{L}{2}\|\delta\|^2.$$

Hint: It may help to use this trick: for any vector $v \in \mathbb{R}^d$, we can always write it as $(v/\|v\|) \cdot \|v\|$, so $v/\|v\|$ is a unit vector.

From Problem 1(b), we have that

$$F(w + \delta) \leq F(w) + \nabla F(w)^\top \delta + \frac{L}{2}\|\delta\|^2.$$

This says that $F(w + \delta)$ is no larger than the quadratic function $G : \mathbb{R}^d \to \mathbb{R}$ of δ,

$$G(\delta) := \frac{L}{2}\|\delta\|^2 + \nabla F(w)^\top \delta + F(w).$$

Compare this to a single-variable quadratic function,

$$ax^2 + bx + c.$$

Problem 1(b) (5 points)

Prove that the minimizer of $G(\delta)$ is the vector

$$\delta^* = -\frac{1}{L}\nabla F(w)$$

using calculus. You may find the matrix calculus identities you used in Problem Set 1 useful. State how this δ^* relates to the definition of gradient descent.

In Problem 1(b), you found the value of δ^* that makes the right-hand side in the upper bound

$$F(w+\delta) \leq F(w) + \nabla F(w)^\top \delta + \frac{L}{2}\|\delta\|^2$$

the smallest. In words, this is the direction δ from w that, when taken, gives the sharpest guarantee on how much $F(w+\delta)$ decreases from $F(w)$. We can use this to finalize our proof of the descent lemma.

Problem 1(c) (5 points)

Finally, prove the descent lemma using your choice of δ^* from Problem 1(b). That is, prove that

$$F\left(w - \frac{1}{L}\nabla F(w)\right) \leq F(w) - \frac{1}{2L}\|\nabla F(w)\|^2.$$

State why this proves the descent lemma.

Problem 2: GD for Linear Regression (35 Points)

In this coding problem, we will be implementing (batch/full) gradient descent for the linear regression problem. Although you will likely never have to do this from scratch again (most major machine learning libraries will have this implemented for you), it'll be instructive to walk through how one might implement this from scratch so you understand the inner workings of such libraries. The skeleton for all the functions mentioned in this problem can be copied/pasted from the file `ps2_skeleton.py`.

Let's recall the linear regression setup from lecture. In linear regression, the input space is $\mathcal{X} = \mathbb{R}^d$, the output/label space is $\mathcal{Y} = \mathbb{R}$, and the action space is $\mathcal{A} = \mathcal{Y} = \mathbb{R}$. We consider the hypothesis class of linear functions:

$$\mathcal{H} := \{x \mapsto w^\top x : w \in \mathbb{R}^d\}.$$

A hypothesis from this class looks like

$$h_w(x) = w^\top x.$$

That is, to predict on a new $x \in \mathbb{R}^d$, we simply take a dot product with w. We measured how badly we did in the linear regression problem on a point (x, y) by the *squared loss*:

$$\ell(h_w(x), y) = (h_w(x) - y)^2 = (w^\top x - y)^2.$$

Because each hypothesis in $\mathcal{H}$ corresponds to exactly one $w \in \mathbb{R}^d$, we will ease up notation and use w by itself to denote the hypothesis $h_w(x) = w^\top x$.

At the end of the day, we care about the *risk* of a hypothesis from this class,

$$R(w) := \mathbb{E}_{(x,y)\sim P_{\mathcal{X}\times\mathcal{Y}}}\left[(w^\top x - y)^2\right].$$

We don't know $P_{\mathcal{X}\times\mathcal{Y}}$ in a typical machine learning problem, but we have a dataset $D_n := \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ that we assume is drawn i.i.d. from $P_{\mathcal{X}\times\mathcal{Y}}$. Our goal, then, is to minimize the *empirical risk*:

$$\hat{R}_n(w) := \frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)} - y^{(i)})^2.$$

Recall from Problem Set 1 that you were able to write this as a *design matrix* $X \in \mathbb{R}^{n\times d}$ and *output vector* $y \in \mathbb{R}^n$ in equivalent matrix-vector form:

$$\hat{R}_n(w) = \frac{1}{n}\|Xw - y\|^2.$$

Before running gradient descent or doing any machine learning, we will take a couple of preliminary preprocessing steps on our data.

Feature normalization. When feature values differ greatly, we can get much slower rates of convergence for gradient-based algorithms. Furthermore, when we start using regularization (in a later problem), features with larger values would be treated as "more important," which is not usually what you want. One common way to approach feature normalization is to perform an affine transformation (i.e. a 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, depending on the scale of that feature.

When using our hypothesis on a validation or test set, we will apply the *same* transformation we did on the training set to the new points in validation or test. It is important that the transformation you use is "learned" on the training set and then applied to the test/validation set! It is possible that some transformed values will lie outside of $[0, 1]$.

Problem 2(a) (3 points)

Find the function `feature_normalization` in the skeleton code and implement the function described in the specification. Your function should take `train` and `test`, which are two 2D `numpy` arrays of size `(num_instances, num_features)` (in our mathematical notation, $n \times d$). It should output the normalized arrays `train_normalized` and `test_normalized`. Note that features that are constant cannot be normalized in this way – your function should discard features that are constant in the training set.
Hint: Using `numpy`'s built-in "broadcasting" feature may be helpful here for writing simpler code.
Submission: Submit the Python code of your implemented function, `feature_normalization`.

Adding the intercept. In lecture, we saw that there is a trick to add a "bias" or intercept term to the hypotheses we are choosing from that just requires adding one more feature from the dataset. Recall that, as written above, the class of linear predictors are just functions of the form:

$$h_w(x) = w^\top x$$

but this only includes lines/planes/hyperplanes that go through the origin. Ideally, we would like to choose from the space of *affine functions* (i.e. linear functions with an intercept):

$$h_w(x) = w^\top x + w_0.$$

The standard way to achieve this is to add an extra dimension to each $x \in \mathbb{R}^d$ that is just the constant 1. Then, we'll actually be solving the problem where $x, w \in \mathbb{R}^{d+1}$. In matrix-vector form, this would mean that you would have a *design matrix* of dimensions $\tilde{X} \in \mathbb{R}^{n \times (d+1)}$ and a weight vector $\tilde{w} \in \mathbb{R}^{d+1}$, while $y \in \mathbb{R}^n$ remains the same dimension.

When you start working with `data.csv` for this problem, don't forget to apply this trick!

Problem 2(b) (2 points)

Let $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$ be the original training inputs for your problem and let $y \in \mathbb{R}^n$ be the output vector. Prove that, by appending a 1 to each $x^{(i)} \in \mathbb{R}^d$ to get

$$\tilde{x}^{(i)} = \begin{pmatrix} x_1^{(i)} & x_2^{(i)} & \ldots & x_d^{(i)} & 1 \end{pmatrix} \in \mathbb{R}^{d+1}$$

and appending w_0 to $w \in \mathbb{R}^d$ to get $\tilde{w} \in \mathbb{R}^{d+1}$, the following are equivalent:

$$\|\tilde{X}\tilde{w} - y\|^2 = \sum_{i=1}^{n} (w^\top x^{(i)} + w_0 - y^{(i)})^2,$$

where $\tilde{X} \in \mathbb{R}^{n \times (d+1)}$ is the design matrix obtained from $x^{(1)}, \ldots, x^{(n)}$ transformed to $\tilde{x}^{(1)}, \ldots, \tilde{x}^{(n)}$ in this way.

Now, we are ready to begin implementing and experimenting with gradient descent on this problem. Recall that when we apply gradient descent to minimize a function, we call that function our *objective function*, $F : \mathbb{R}^d \to \mathbb{R}$. In our case, our objective function is the empirical risk on the fixed dataset:

$$F(w) = \hat{R}_n(w) = \frac{1}{n}\|Xw - y\|^2.$$

For this problem, we will be keeping the $1/n$ factor so we can make sure we are minimizing the empirical risk itself (and not just the sum of losses).

First, we'll write out exactly what gradient descent is for this objective function before writing code. Note that our squared loss objective function is particularly easy to deal with and we could write out its gradient (and Hessian) by hand by just applying the rules of calculus. Later, when we perform gradient descent on neural networks, it'll be much harder to write down an expression in this way.

From lecture, we saw an approximate heuristic argument using the definition of the multivariate derivative for why the gradient descent update rule might be a good step direction to take.

Problem 2(c) (2 points)

Consider a general differentiable objective function $F : \mathbb{R}^d \to \mathbb{R}$ (not necessarily the squared loss empirical risk we are considering). Use the approximation argument from lecture to show that stepping from $w^{(t)}$ to $w^{(t+1)} = w^{(t)} + \eta\delta$ where $\eta > 0$ and $\delta \in \mathbb{R}^d$ is some vector follows the approximation:

$$F(w^{(t+1)}) - F(w^{(t)}) \approx \eta \nabla F(w^{(t)})\delta.$$

From Problem Set 1, you should be familiar with how to take a derivative of the particular objective function we are working with, and, thus, you can write down what gradient descent looks like for our problem.

Problem 2(d) (3 points)

Write down the expression for the gradient descent update rule for our objective function:

$$F(w) = \frac{1}{n}\|Xw - y\|^2.$$

You'll need to take the gradient $\nabla_w F(w)$ (you may use the hints from Problem Set 1 for this problem). Don't forget the $1/n$ factor!

We will now begin experimenting and writing code. The following two problems will calculate your objective function itself and the gradient, which we already have written down by hand.

Problem 2(e) (2 points)

Modify the function `compute_square_loss` to compute our squared loss objective $F(w)$ for a given $w \in \mathbb{R}^d$. It should take in $X \in \mathbb{R}^{n \times d}$, y, and $w \in \mathbb{R}^d$ and output a scalar $F(w) \in \mathbb{R}$. You may want to create a small dataset for which you can compute $F(w)$ by hand and verify that `compute_square_loss` outputs the correct value. You may use your code from Problem Set 1; make sure you keep the $1/n$ factor and it outputs correctly!
Submission: Submit the Python code of your implemented function, `compute_square_loss`.

Problem 2(f) (3 points)

Modify the function `compute_square_loss_gradient` to compute our squared loss gradient $\nabla_w F(w)$ for a given $w \in \mathbb{R}^d$. It should take in $X \in \mathbb{R}^{n \times d}$, y, and $w \in \mathbb{R}^d$ and output a vector $\nabla_w F(w) \in \mathbb{R}^d$. You may want to create a small dataset for which you can compute $\nabla_w F(w)$ by hand and verify that `compute_square_loss_gradient` outputs the correct value.
Submission: Submit the Python code of your implemented function, `compute_square_loss_gradient`.

For many optimization problems, coding the gradient up correctly can be tricky. Luckily, there is a nice way to numerically check the gradient calculation. If $F : \mathbb{R}^d \to \mathbb{R}$ is differentiable, then for any vector $\delta \in \mathbb{R}^d$, the directional derivative of F at w in the direction δ is

given by[1]

$$\lim_{\epsilon \to 0} \frac{F(w + \epsilon\delta) - F(w - \epsilon\delta)}{2\epsilon}.$$

We can approximate this directional derivative by choosing a small value of $\epsilon > 0$ 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 $\delta = (1, 0, 0, \ldots, 0)$ to get the first component of the gradient. Then take $\delta = (0, 1, 0, \ldots, 0)$ to get the second component. And so on. See http://deeplearning.stanford.edu/tutorial/supervised/DebuggingGradientChecking/ for details.

Problem 2(g) (5 points)

Complete the function `grad_checker` according to the documentation given. It should take as parameters X, y, w, a function that computes the objective function $F(u)$ for any $u \in \mathbb{R}^d$ (e.g. `compute_square_loss`) and a function that computes the gradient of the objective function $\nabla F(u)$ for any $u \in \mathbb{R}^d$ (e.g. `compute_square_loss_gradient`).
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.
Submission: Submit the Python code of your implemented function, `grad_checker`.

At the end of the skeleton code, the data is loaded, split into a training and test set, and normalized. Note that the skeleton code also does the "bias 1" trick for you, so, without loss of generality, just assume that dimension d includes the extra intercept dimension. We'll now finish the job of running gradient descent on the training set. Later on, we'll plot the results against the SGD results.

Problem 2(h) (5 points)

Complete the function `batch_gradient_descent` to implement (full batch) gradient descent for this problem. It should take in $X \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, with optional parameters for the step size $\eta > 0$, the number of steps to stop at, and whether to use the `grad_checker` function from Problem 2(g). It should output two arrays: one is for the history of the w values $w^{(1)}, \ldots, w^{(T)}$, and one is for the history of the objective values $F(w^{(1)}), \ldots, F(w^{(T)})$.
You should use the functions you implemented in the previous problems in this function. If `grad_checker` fails, stop the run of gradient descent and throw an exception of your choice.
Submission: Submit the Python code of your implemented function, `batch_gradient_descent`.

[1] Of course, it is also given by the more standard definition of directional derivative, $\lim_{\epsilon \to 0} \frac{1}{\epsilon}[F(w + \epsilon\delta) - F(w)]$. The form given gives a better approximation to the derivative when we are using small (but not infinitesimally small) ϵ.

Problem 2(i) (5 points)

Now we will experiment with the step size $\eta > 0$. Note that if the step size is too large, gradient descent may not converge. Starting with a step-size of 0.1, try the following six step sizes:

$$\eta \in \{0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}.$$

You can try more if you're curious.
For each step size, plot the average square loss on the training data $F(w)$ as a function of the number of steps for each step size. Your plot should have a curve for each step size that converges, with number of steps on the x-axis and $F(w)$ on the y-axis. Do not plot the curves for the step sizes that do not converge. Briefly summarize your findings.
Submission: Submit only your plot for this problem.

It turns out that our squared loss objective

$$F(w) = \frac{1}{n}\|Xw - y\|^2$$

is a convenient one to find the smoothness parameter of. Recall the *descent lemma* from lecture that you prove in Problem 1. This lemma tells you which step sizes are *guaranteed* to make your objective function smaller for twice-differentiable functions. Thankfully, the F we are dealing with is twice-differentiable and has a nicely interpretable second derivative.

Problem 2(j) (5 points)

Using the definition of L-smooth from lecture that applies to the Hessian of a function, what is a bound on the smoothness parameter of the $F(w)$? You'll need to find the Hessian of F, i.e. $\nabla^2 F(w) \in \mathbb{R}^{d \times d}$. Using this information, what step sizes does the descent lemma tell us *must* decrease the objective?
Finally, fill out the function `compute_l_smoothness_constant` which only takes the matrix of features $X \in \mathbb{R}^{d \times d}$ and outputs a scalar $L \in \mathbb{R}$. You may find `np.linalg.eigvals` useful. Apply `compute_l_smoothness_constant` to your dataset that you experimented with in Problem 2(h). Briefly discuss how this result relates to what you saw in Problem 2(h). Remember that the theorem only says something about $\eta \leq 1/L$.
Submission: Submit Python code for `compute_l_smoothness_constant` along with the written answers to the questions above.

Problem 3: Ridge Regression (25 Points)

We are again in the linear regression setup with squared loss described in lecture and in Problem 2. Recall from class that one of our main issues in machine learning is ensuring the *estimation error* and *approximation error* for our problem are both small. For a hypothesis class $\mathcal{H}$, let the empirical risk minimizer (ERM) be

$$\hat{h}_n \in \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(h(x^{(i)}), y^{(i)}))$$

and let the risk minimizer be

$$h^*_{\mathcal{H}} \in \arg\min_{h \in \mathcal{H}} \mathbb{E}[\ell(h(x), y)].$$

The *estimation error* is the difference between the risks of the ERM and the risk minimizer

$$R(\hat{h}_n) - R(h^*_{\mathcal{H}}).$$

Typically, we can decrease this quantity by getting more data. If that's not an option, then a strategy that sometimes decreases this quantity is decreasing the "size" or "complexity" of $\mathcal{H}$. Intuitively, we might expect this to control this gap because limiting the options in $\mathcal{H}$ for our ERM may exclude hypotheses that overfit to the noise in the training data. Remember: for most problems there can be many ERMs (so a "large" $\mathcal{H}$ might allow you to choose one that won't generalize well)! *Regularization* is a general-purpose term that describes ways to reduce model complexity.

We will hope that decreasing the "complexity" of $\mathcal{H}$ does not hurt our approximation error too much (or at all). The approximation error is the difference in risk between the best hypothesis we can find in our class $\mathcal{H}$ and the best possible function anyone could ever produce for our problem (the Bayes hypothesis):

$$R(h^*_{\mathcal{H}}) - R(h^*).$$

In this problem, we will employ ℓ_2-regularization (otherwise known as *ridge regression*) of linear regression as a tactic to decrease estimation error quicker than we can increase approximation error. That is, we will use regularization to try to prevent overfitting.

Recall from class that the ℓ_2-regularized linear regression (a.k.a. *ridge regression*) objective function is

$$F(w) = \frac{1}{n} \sum_{i=1}^{n} (w^\top x^{(i)} - y^{(i)})^2 + \lambda \|w\|^2,$$

where $\lambda \geq 0$ is known as the *regularization parameter*. We will assume that the above formulation applies the "dummy 1" trick so the bias term w_0 is getting regularized as well (we will address this later). Without loss of generality, we'll just assume that $w \in \mathbb{R}^d$ includes w_0 already.

Just as in Problem Set 1, we can compute the closed form solution for ridge regression. We will prove another useful lemma from linear algebra (this should be compared to why we needed the lemma $\text{rank}(X) = \text{rank}(X^\top X)$ in un-regularized linear regression). One way to interpret the following lemma is that perturbing a matrix with λI for positive λ gets rid of linear dependencies.

Problem 3(a) (2 points)

Prove that, for any matrix $X \in \mathbb{R}^{n \times d}$ and $\lambda \in \mathbb{R}$, the matrix $X^\top X + \lambda I$ is always invertible if $\lambda > 0$.
Hint: You may find that analyzing $u^\top (X^\top X + \lambda I)u$ for an arbitrary nonzero $u \in \mathbb{R}^d$ is useful. You may also use the fact that any positive definite matrix is invertible.

Using Problem 3(a), we can prove the closed form solution to ridge regression in a similar way to how we showed the closed form solution to unregularized linear regression in Problem Set 1.

Problem 3(b) (3 points)

Let $\lambda > 0$. Using Problem 3(a), prove that *for any* $X \in \mathbb{R}^{n \times d}$, the closed form solution to the minimization problem

$$\hat{w} \in \arg\min_{w \in \mathbb{R}^d} \sum_{i=1}^{n} (w^\top x^{(i)} - y^{(i)})^2 + \lambda \|w\|^2$$

(where we took out the $1/n$ for convenience) is given by

$$\hat{w} = (X^\top X + \lambda I)^{-1} X^\top y.$$

Don't forget to verify using calculus that this is indeed the minimizer. You will need to use the matrix-vector form of the objective to solve this.
Compare this to the closed form solution of unregularized linear regression. What is the difference with the conditions on X you need for ridge regression vs. unregularized linear regression?

Problem 3(a) and Problem 3(b) show that just as in (unregularized) linear regression, finding a closed form solution to the minimization problem can be done via calculus. As you now know, however, in machine learning, we like applying our bread and butter algorithm, *gradient descent*, to minimize differentiable objectives.

Problem 3(c) (2 points)

Write down the gradient descent step updating $w^{(t)}$ for ridge regression (given by the $F(w)$ above) in matrix-vector form. Don't forget the $1/n$ term in this case!

Problem 3(d) (3 points)

Implement `compute_regularized_square_loss_gradient`. This function should take X, y, w and the regularization parameter λ. It should output the gradient, $\nabla F(w) \in \mathbb{R}^d$.
Submission: Submit your Python code for `compute_regularized_square_loss_gradient`.

Now that we have a function that computes the gradient, we can implement gradient descent just as we did in Problem 2.

Problem 3(e) (2 points)

Implement `regularized_grad_descent` for ridge regression. It should take in $X \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$, with optional parameters for the step size $\eta > 0$, the number of steps to stop at, and the regularization parameter λ. It should output two arrays: one is for the history of the w values $w^{(1)}, \ldots, w^{(T)}$, and one is for the history of the objective values $F(w^{(1)}), \ldots, F(w^{(T)})$
Submission: Submit your Python code for `regularized_grad_descent`.

For regression problems, you may prefer to leave the bias term unregularized. One approach is to change $F(w)$ 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 appended to the x.

Problem 3(f) (3 points)

Explain why making B large decreases the effective regularization on the bias term. Explain how we can make the regularization as weak as we like on the bias term (though not exactly zero). A formal proof is not needed; just consider what happens when one varies B.

Up until this point, we have only considered the *empirical risk*, or the loss of our hypotheses on our training set. When performing optimization, this is the quantity we focus on making small. However, our *ultimate goal* is, of course, performing well on unseen data, or minimizing our true risk. With squared loss and linear regression, this is:

$$R(w) = \mathbb{E}_{(x,y)\sim P_{\mathcal{X}\times\mathcal{Y}}}[(w^\top x - y)^2]$$

Of course, we don't ever know what the true distribution $P_{\mathcal{X}\times\mathcal{Y}}$ is in a machine learning problem, so we estimate the true risk with an i.i.d. *test set* that is separate from our training set. Denoting our test set as $(\tilde{x}^{(1)}, \tilde{y}^{(1)}), \ldots, (\tilde{x}^{(m)}, \tilde{y}^{(m)})$, we estimate

$$R(w) = \mathbb{E}_{(x,y)\sim P_{\mathcal{X}\times\mathcal{Y}}}[(w^\top x - y)^2] \approx \frac{1}{m}\sum_{j=1}^{m}(w^\top \tilde{x}^{(j)} - \tilde{y}^{(j)}).$$

Remember that whenever we are calculating the test or validation error of w, we are approximating the true risk with an separate i.i.d. sample.

Problem 3(g) (5 points)

Now fix $B = 1$ (it is fine to regularize the bias term in this problem) and $\eta = 0.05$. Find the $\hat{w}_\lambda$ that minimizes $F(w)$ for a range of λ using gradient descent. Plot the average square loss on the training set and the test set (the original, un-regularized objectives) as a function of λ (your plot should have two curves) for λ in `np.logspace(-7, -1, num=30)`. You should have $\log(\lambda)$ on your x-axis rather than λ.

Your goal is to find λ that gives the minimum average square loss on the test set. Report that optimal λ. It's hard to predict what λ should be, so you should start your search very broadly, looking over several orders of magnitude. Begin with the λ in `np.logspace(-7, -1, num=30)`. Once you can see a range that works better, keep zooming in. If you'd like, you can use `sklearn` to help with this hyperparameter search.

Submission: The plot of $\log(\lambda)$ vs. average square loss for training and test *and* your optimal λ value from experimenting. We will accept any λ that is around the same order of magnitude (within 10 times) the optimal λ. Update: Previously, the problem read $\eta = 0.5$, but you should be using the default value in the skeleton code, $\eta = 0.05$.

Problem 3(h) (5 points)

Which $\hat{w}_\lambda$ would you select for deployment? You do not need to report the coefficients; just state which $\hat{w}_\lambda$ you would then report to your boss after this experiment as the most likely to do well on future data from the same distribution. This is not a trick question!

Problem 4: Stochastic Gradient Descent (25 Points)

When the training 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. When the objective function takes the form of an average of many values, such as

$$F(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$$

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

In machine learning applications, each $f_i(w)$ would be the loss on the ith example. 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. 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.

Problem 4(a) (3 points)

Show that the objective function

$$F(w) = \frac{1}{n}\sum_{i=1}^{n} \left(w^\top x^{(i)} - y^{(i)}\right)^2 + \lambda \|w\|^2$$

can be written in the form $F(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w)$ by giving an expression for $f_i(w)$ that makes the two expressions equivalent. This tells us we can perform SGD on our regularized linear regression objective.

Recall from probability and statistics that an *unbiased estimator* is an estimator (a rule for calculating a quantity given observed data) whose expectation is the true value of the quantity being estimated. In the case of SGD, our "observed data" are the draws $i \in [n]$ from the uniform distribution over $\{1, \ldots, n\}$ at each step, which tells us a particular stochastic gradient $\nabla f_i(w)$ to draw.

Problem 4(b) (5 points)

Prove that the stochastic gradient $\nabla f_i(w)$ for i chosen uniformly at random from $\{1, \ldots, n\}$ is an unbiased estimator of $\nabla F(w)$. That is, show that, for any w,

$$\mathbb{E}[\nabla f_i(w)] = \nabla F(w).$$

It will be helpful to think about what the expectation on the left hand side is really over. *Hint: It will be notationally easier to prove this for a general* $F(w) = \frac{1}{n} f_i(w)$ *than for the ridge regression objective. Of course, proving this in general means that the ridge regression estimator is also unbiased, because ridge regression is a special case, as Problem 4(a) shows.*

Problem 4(c) (2 points)

Write down the update rule for $w^{(t)}$ in SGD for the ridge regression objective function.

Theoretically, we typically analyze SGD thinking of each stochastic gradient as sampled uniformly at random as above. In practice, SGD is done in *epochs*. At the start of an epoch, one typically shuffles the data. Then, an epoch goes through each of the n points in the shuffled order. Therefore, one epoch goes touches exactly as much data as a single full gradient descent step. On the next epoch, one typically reshuffles the data.

Problem 4(d) (10 points)

Implement `stochastic_grad_descent`. Make sure you implement the epochs as stated above. To shuffle data, `np.random.permutation` may help. Note that the skeleton code takes in three options for `eta` for step t during gradient descent:

- If `eta` is a float, then gradient descent uses that float as the step size for every iteration.
- If `eta` is the string `"1/sqrt(t)"`, then use $\eta_t = 1/\sqrt{t}$.
- If `eta` is the string `"1/t"`, then use $\eta_t = 1/t$.

Although SGD is done in epochs, each gradient step increases t by one. For instance, if the dataset has $n = 100$ points and you are on the fifth datapoint of the second epoch, then $t = 105$.
Also, make sure that the `loss_hist` records the loss of the regularized objective. We will be investigating the convergence rate of the optimization algorithm, so we care about the objective function itself (which includes our regularization term), not our downstream metric of risk.
We will not be implementing minibatches for this problem set, but, as an optional exercise, you can generalize this code to accept minibatches instead of stochastic gradients that are merely a single point.
Submission: Submit your Python code for `stochastic_grad_descent`

Finally, we will experiment with the convergence rate of SGD with different learning rates. We will not be evaluating the risk on the test set; we are primarily concerned with the performance of just the optimization problem itself.

Problem 4(e) (5 points)

Set $\lambda = 10^{-2}$ in ridge regression. Use SGD for 1000 epochs with the following five step sizes:

1. Fixed step size $\eta = 0.005$.
2. Fixed step size $\eta = 0.01$.
3. Step size $\eta_t = 0.1/\sqrt{t}$.
4. Step size $\eta_t = 0.1/t$.

You can modify the code in `stochastic_grad_descent` to make the step sizes with $c/\sqrt{t}$ or c/t work however you'd like. For each step size rule, plot the value of the objective function on the y-axis as a function of the epoch (or step number, if you prefer). The y-axis should be on a logarithmic scale. There should be four curves on the plot. How do the results compare? It's possible that some of these rules will not converge.
You are encouraged to experiment and try out other learning rates to get a feel tweaking learning rates for SGD.
Submission: Just submit the plot from the experiment.