CPSC 540 Assignment 2 (due February 6)

Large-Scale Machine Learning

0 Unofficial Course Evaluation

To help improve the course as we go along, or to suggest of how things could be done differently, please fill out the survey here:
`https://survey.ubc.ca/surveys/37-7974b781afc90962f53d98c2749/cpsc-540-informal-course-evaluation`

1 Convergence Rates

1.1 Gradient Descent

For minimizing a function f, in class we showed that if ∇f is Lipschitz continuous and f is strongly-convex then gradient descent iterations,

$$x^{t+1} = x^t - \alpha_t \nabla f(x^t),$$

with a step-size of $\alpha_t = 1/L$ satisfy

$$f(x^t) - f(x^*) = O(\rho^t),$$

for some $\rho < 1$ (we call this a linear convergence rate). In this question you'll show some related properties.

1. The rate above is in terms of the function value $f(x^t)$, but we might also be interested in the convergence of the iterates x^t to x^*. Show that if f is differentiable and strongly-convex then a convergence rate of $O(\rho^t)$ in terms of the function values implies that the iterations have a convergence rate of

$$\|x^t - x^*\| = O(\rho^{t/2}).$$

SOLUTION

Since f is differentiable and strongly convex we have, for any x, y

$$f(y) \geq f(x) + \nabla f(x)^T(y - x) + \frac{\mu}{2}\|y - x\|_2^2$$

In particular, substituting for x^* and x^t it follows that

$$f(x^t) \geq f(x^*) + \underbrace{\nabla f(x^*)^T}_{=0}(x^t - x^*) + \frac{\mu}{2}\|x^t - x^*\|_2^2 = f(x^*) + \frac{\mu}{2}\|x^t - x^*\|_2^2$$

Hence, using $f(x^t) - f(x^*) = O(\rho^t)$,

$$O(\rho^t) = f(x^t) - f(x^*) \geq \frac{\mu}{2}\|x^t - x^*\|_2^2 \quad \Longrightarrow \quad \|x^t - x^*\|_2 \leq O(\rho^{t/2}).$$

2. Consider using a constant step-size $\alpha_t = \alpha$ for some positive constant $\alpha < 2/L$. Show that gradient descent converges linearly under this alternate step-size (you can use the descent lemma).

   SOLUTION

   Fix $\alpha < 2/L$ and define the iterates

   $$x^{t+1} := x^t - \alpha \nabla f(x^t).$$

   The goal is to show that $f(x^t) - f(x^*) = O(\rho^t)$ for $\rho < 1$. By the descent lemma,

   $$f(y) - f(x) \leq \nabla f(x)^T (y - x) + \frac{L}{2}\|y - x\|_2^2.$$

   Hence for $x := x^t$ and $y := x^{t+1}$ it follows that

   $$\begin{aligned} f(x^{t+1}) - f(x^t) &\leq \nabla f(x^t)^T (x^{t+1} - x^t) + \frac{L}{2}\|x^{t+1} - x^t\|_2^2 \\ &= \left(\frac{\alpha^2 L}{2} - \alpha\right)\|\nabla f(x^t)\|_2^2 \\ &< -\alpha\|\nabla f(x^t)\|_2^2 + \alpha\|\nabla f(x^t)\|_2^2 = 0 \end{aligned}$$

3. In practice we typically don't L. A common strategy in this setting is to start with some small guess L^0 that we know is smaller than the true L (usually we take $L = 1$). On each iteration t, we initialize with $L^t = L^{t-1}$ and we check the inequality

   $$f\left(x^t - \frac{1}{L^t}\nabla f(x^t)\right) \leq f(x^t) - \frac{1}{L^t}\|\nabla f(x^t)\|^2.$$

   If this is not satisfied, we double L^t and test it again. This continues until we have an L^t satisfying the inequality. Show that gradient descent with $\alpha_t = 1/L^t$ defined in this way has a linear convergence rate of

   $$f(x^t) - f(x^*) \leq \left(1 - \frac{\mu}{2L}\right)[f(x^0) - f(x^*).$$

   Hint: if a function is L-Lipschitz continuous that it is also L'-Lipschitz continuous for any $L' \geq L$.

4. Describe a condition under which the step-sizes in the previous question would give a faster rate than $\rho = (1 - \mu/L)$.

1.2 Sign-Based Gradient Descent

In some situations it might be hard to accurately compute the elements of the gradient, but we might have access to the sign of the gradient. For this setting, consider a sign-based gradient descent algorithm of the form

$$x^{t+1} = x^t - \frac{\|\nabla f(x^t)\|_1}{L}\text{sign}(\nabla f(x^t)),$$

where we define the sign function element-wise as

$$\text{sign}(x_j) = \begin{cases} +1 & x_j > 0 \\ 0 & x_j = 0 \\ -1 & x_j < 0 \end{cases}.$$

Consider an f that is strongly-convex and is Lipschitz continuous in the ∞-norm, meaning that

$$f(y) \leq f(x) + \nabla f(x)^T (y - x) + \frac{L_\infty}{2}\|y - x\|_\infty^2,$$

for all y and x and some L_∞.

1. Show that the sign-based gradient descent method satisfies

$$f(x^{t+1}) - f(x^*) \leq \left(1 - \frac{\mu}{L_\infty}\right)[f(x^t) - f(x^*)].$$

2. To compare this rate to the rate of gradient descent, we need to know the relationship between the usual Lipschitz constant L (in the 2-norm) and L_∞. Show that the relationship between these constants is

$$L_\infty \leq L \leq dL_\infty.$$

1.3 Block Coordinate Descent

Consider a problem it makes sense to partition our variables into k disjoint 'blocks'

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix},$$

each of size d/k. Assume that f is *strongly-convex*, and *blockwise strongly-smooth*,

$$\nabla^2 f(w) \succeq \mu I, \quad \nabla^2_{bb} f(w) \preceq LI,$$

for all w and all blocks b. Consider a *block coordinate descent* algorithm where we use the iteration

$$x^{t+1} = x^t - \frac{1}{L}(\nabla f(x^t) \circ e_{b_t}),$$

where b_t is the block we choose on iteration t, e_b is vector of zeros with ones at the locations of block b, and $\circ$ means element-wise multiplication of the two vectors. (It's like coordinate descent except we're updating d/k variables instead of just one.)

1. Assume that we pick a random block on each iteration, $p(b_t = b) = 1/k$. Show that this method satisfies

$$\mathbb{E}[f(x^{t+1})] - f(x^*) \leq \left(1 - \frac{\mu}{Lk}\right)[f(x^t) - f(x^*)].$$

2. Assume that each block b has its own strong-smoothness constant L_b,

$$\nabla^2_{bb} f(w) \preceq L_b I,$$

so that the strong-smoothness constant from part 1 is given by $L = \max_b\{L_b\}$. Show that if we sample the blocks proportional to L_b, $p(b_t = b) = \frac{L_b}{\sum_{b'} L_{b'}}$ and we use a larger step-size of $1/L_{b_t}$, then we obtain a faster convergence rate provided that some $L_b \neq L$.

2 Large-Scale Algorithms

2.1 Coordinate Optimization

The function *example_logistic* loads a dataset and tries to fit an L2-regularized logistic regression model using coordinate optimization. Unfortunately, if we use L_f as the Lipschitz constant of ∇f, the runtime of this procedure is $O(d^3 + nd^2\frac{L_f}{\mu}\log(1/\epsilon))$. This comes from spending $O(d^3)$ computing L_f, having an iteration cost of $O(nd)$, and requiring a $O(d\frac{L_f}{\mu}\log(1/\epsilon))$ iterations. This non-ideal runtime is also reflected in practice: the algorithm's iterations are relatively slow and even after 500 "passes" through the data it isn't particularly close to the optimal function value.

1. Modify this code so that the runtime of the algorithm is $O(nd\frac{L_c}{\mu}\log(1/\epsilon))$, where L_c is the Lipschitz constant of *all* partial derivatives $\nabla_i f$. You can do this by modifying the iterations so they have a cost $O(n)$ instead of $O(nd)$, and instead of using a step-size of $1/L_f$ they use a step-size of $1/L_c$ (which is given by $\frac{1}{4}\max_j\{\|x_j\|^2\} + \lambda$). Hand in your code and report the final function value and total time.
2. To further improve the performance, make a new version of the code which samples the variable to update j_t proportional to the individual Lipschitz constants L_j of the coordinates, and use a step-size of $1/L_{j_t}$. You can use the function *sampleDiscrete* to sample from discrete distribution given the probability mass function. Hand in your code, and report the final function value as well as the number of passes.
3. Report the number of passes the algorithm takes as well as the final function value if you use *uniform sampling* but use a step-size of $1/L_{j_t}$.
4. Suppose that when we use a step-size of $1/L_{j_t}$, we see that uniform sampling outperforms Lipschitz sampling. Why would this be consistent with the bounds we stated in class?

SOLUTION

We include the output of the default run (as base-line) as well as the three requested modifications, labeled "original", and "coordinateX" where $X \in \{1, 2, 3\}$, respectively.

```
original:
Passes = 500, function = 1.4735e+02, change = 0.0003
Elapsed time is 17.793978 seconds.

coordinate1:
Passes = 500, function = 1.4723e+02, change = 0.0005
Elapsed time is 3.021629 seconds.

coordinate2:
Passes = 291, function = 1.4330e+02, change = 0.0001
Parameters changed by less than progTol on pass
Elapsed time is 2.132973 seconds.

coordinate3:
Passes = 136, function = 1.4091e+02, change = 0.0001
Parameters changed by less than progTol on pass
Elapsed time is 0.823345 seconds.
```

To answer the latter question, note that where uniform sampling is used — $\forall j \in [d],\ p(j) = d^{-1}$ — we achieve the guaranteed progress bound

$$\mathbb{E}(f(x^{t+1})) \leq f(x^t) - \frac{1}{2dL}\|\nabla f(x^t)\|^2$$

where $[d] := \{1, \ldots, d\}$. Correspondingly, for Lipschitz sampling, the pmf is given by

$$\forall j \in [d], \quad p(j) = \frac{L_j}{\sum_{j\in[d]} L_j} =: \frac{L_j}{d\overline{L}}$$

where $d\overline{L}$ is the normalizing constant for the pmf, with $\overline{L}$ the arithmetic mean of the coordinate-wise Lipschitz constants L_j. Hence, the guaranteed progress bound for Lipschitz sampling is

$$\begin{aligned}\mathbb{E}f(x^{t+1}) &\leq \mathbb{E}\big(f(x^t) - \frac{1}{2L}|\nabla_{j_t} f(x^t)|^2\big) = \sum_{j=1}^{d} \frac{L_{j_t}}{d\overline{L}}\big(f(x^t) - \frac{1}{2L}|\nabla_j f(x^t)|^2\big) \\ &= f(x^t) - \frac{1}{2dL}\sum_{j\in d}\frac{L_{j_t}}{\overline{L}}|\nabla_{j_t} f(x^t)|^2\end{aligned}$$

In particular, uniform sampling is on average preferred to Lipschitz sampling if

$$\langle \mathbf{1}, |\nabla_{j_t} f(x^t)|^2\rangle = \|\nabla f(x^t)\|^2 > \sum_{j_t\in[d]} \frac{L_{j_t}}{\overline{L}}|\nabla_{j_t} f(x^t)|^2 = \langle \frac{L_{j_t}}{\overline{L}}, |\nabla_{j_t} f(x^t)|^2\rangle.$$

More generally, it is desirable to have a sampling scheme which is [in the above sense] well-aligned with the squared modulus of the j_t-gradient vector $\big(|\nabla_{j_t} f(x^t)|^2\big)$.

2.2 Proximal-Gradient

If you run the demo *example_group*, it will load a dataset and fit a multi-class logistic regression (softmax) classifier. This dataset is actually *linearly-separable*, so there exists a set of weights W that can perfectly classify the training data (though it may be difficult to find a W that perfectly classifiers the validation data). However, 90% of the columns of X are irrelevant. Because of this issue, when you run the demo you find that the training error is 0 while the test error is something like 0.2980.

1. Write a new function, *softmaxClassifierL2*, that fits a multi-class logistic regression model with L2-regularization (this only involves modifying the objective function). Hand in the modified loss function and report the best validation error achievable with $\lambda = 10$ (which is best value among powers to 10). Also report the number of non-zero parameters in the model and the number of original features that the model uses.

2. While L2-regularization reduces overfitting a bit, it still uses all the variables even though 90% of them are irrelevant. In situations like this, L1-regularization may be more suitable. Write a new function, *softmaxClassifierL1*, that fits a multi-class logistic regression model with L1-regularization. You can use the function *proxGradL1*, which minimizes the sum of a differentiable function and an L1-regularization term. Report the number of non-zero parameters in the model and the number of original features that the model uses.

3. L1-regularization achieves sparsity in the *model parameters*, but in this dataset it's actually the *original features* that are irrelevant. We can encourage sparsity in the original features by using *group* L1-regularization. Write a new function, *proxGradGroupL1*, to allow (disjoint) *group* L1-regularization. Use this within a new function, *softmaxClassiferGL1*, to fit a group L1-regularized multi-class logistic

regression model (where *rows* of W are grouped together and we use the L2-norm of the groups). Hand in both modified functions (*softmaxClassifierGL1* and *proxGradGroupL1*) and report the validation error achieved with $\lambda = 10$. Also report the number of non-zero parameters in the model and the number of original features that the model uses.

2.3 Stochastic Gradient

If you run the demo *example_stochastic*, it will load a dataset and try to fit an L2-regularized logistic regression model using 10 "passes" of stochastic gradient using the step-size of $\alpha_t = 1/\lambda t$ that is suggested in many theory papers. Note that the demo is quite slow as Matlab doesn't do well with 'for' loops, but if you implemented this in C this would be very fast even though there are 50,000 training examples.

Unfortunately, even if we ignore the Matlab-slowness, the performance of this stochastic gradient method is atrocious. It often goes to areas of the parameter space with the objective function overflows and the final value is usually in the range of something like $6.5 - 7.5 \times 10^4$. This is quite far from the solution of 2.7068×10^4 and is even worse than just choosing $w = 0$ which gives 3.5×10^4. (This is unlike gradient descent and coordinate optimization, which never increase the objective function.)

1. Although $\alpha_t = 1/\lambda$ gives the best possible convergence rate in the worst case, in practice it's typically horrible (as we're not usually opitmizing the hardest possible λ-strongly convex function). Experiment with different choices of step-size to see if you can get better performance. Report the step-size that you found gave the best performance, and the objective function value obtained by this strategy for one run.
2. Besides tuning the step-size, another strategy that often improves the performance is using a (possibly-weighted) average of the iterations w^t. Explore whether this strategy can improve performance. Report the performance with an averaging strategy, and the objective function value obtained by this strategy for one run. (Note that the best step-size sequence with averaging might be different than without averaging.)
3. A popular variation on stochastic is AdaGrad, which uses the iteration

$$w^{t+1} = w^t - \alpha_t D_t \nabla f(x^t),$$

   where the element in position (i, i) of the diagonal matrix D_t is given by $1/\sqrt{\delta + \sum_{t=0}^{t} \nabla f(x^t)}$ (and we don't average the steps). Implement this algorithm and experiment with the tuning parameters α_t and δ. Hand in your code as well as the best step-size sequence you found and again report the performance for one run.
4. Impelement the SAG algorithm with a step-size of $1/L$, where the L is the maximum Lipschitz constant across the training examples ($L = 0.25 \max_i\{\|x^i\|^2\} + \lambda$). Hand in your code and again report the performance for one run.

3 Kernels and Duality

3.1 Fenchel Duality

Recall that the Fenchel dual for the primal problem

$$P(w) = f(Xw) + g(w),$$

is the dual problem

$$D(z) = -f^*(-z) - g^*(X^T z),$$

or if we re-parameterize in terms of $-z$:

$$D(z) = -f^*(z) - g^*(-X^T z), \tag{1}$$

where f^* and g^* are the convex conjugates. Convex conjugates are discussed in Section 3.3 of Boyd and Vandenberghe (`http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf`). Read this, then derive the Fenchel dual for the following problems:

1. $P(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w\|^2$ (L2-regularized least squares)
2. $P(w) = \|Xw - y\|_1 + \lambda\|w\|_1$ (robust regression with L1-regularization)
3. $P(w) = \sum_{i=1}^N \log(1 + \exp(-y^i w^T x^i)) + \frac{\lambda}{2}\|w\|^2$ (regularized maximum entropy)

Hint: Don't try to take the Lagrangian dual, a generic strategy to compute the special case of Fenchel duals is as follows:

- Determine X, f, and g to put the problem into the primal format.
- Determine the form of f^* and g^* (note that A here is not relevant).
- Evaluate f^* at $-z$ and g^* at $X^T z$ to get the final form.

For a differentiable f, you can often solve for the value achieving the sup inside of $f^*(v)$ by taking the gradient of $(x^T v - f(x))$ and setting it to zero (keeping in mind whether there are values of v where the sup might be infinity). Example 3.26 in the book gives the convex conjugate in the case where f is a norm. Section 3.3.2 of the book shows how the convex conjugate changes if you scale a function and/or compose a function with an affine transformation. For parts 1 and 2, the X in the primal will just be the data matrix X. But for part 3, it will be easier if you define X as a matrix with row i is given by $y^i x^i$. For part 3 you'll want to use $f(v) = \sum_{i=1}^n \log(1 + \exp(v_i))$, which is a *separable* function (meaning that you can optimize each z_i independently).

Solution

1. Define $f(\omega) := \frac{1}{2}\|\omega - y\|_2^2$ and $g(\omega) := \frac{\lambda}{2}\|\omega\|_2^2$. Since f and $\langle\cdot,\cdot\rangle$ are smooth, the optimal value for ω can be computed coordinate-wise as the solution of

   $$\partial_{\omega_j}\big(\langle\omega, z\rangle - f(\omega)\big) = z_j + y_j - \omega_j = 0.$$

   Hence, since the argument of ∂_{ω_j} is concave, the argument is maximal for $\omega = \omega^* := z_j + y_j$ whence

   $$f^*(z) = \langle y + z, z\rangle - \frac{1}{2}\|z\|_2^2 = \langle y, z\rangle + \frac{1}{2}\|z\|_2^2$$

   Similarly, $\partial_{\omega_j}\big(\langle\omega, z\rangle - \frac{\lambda}{2}\|\omega\|_2^2\big) = z_j - \lambda\omega_j = 0$, implying that $\omega = \omega^* := z/\lambda$. Therefore,

   $$g^*(z) = \frac{1}{2\lambda}\|z\|_2^2$$

   Therefore,

   $$D(z) = -f^*(-z) - g^*(X^T z) = \langle y, z\rangle - \frac{1}{2}\|z\|_2^2 - \frac{1}{2\lambda}\|X^T z\|_2^2.$$

2. Define $f(\omega) := \|\omega - y\|_1,\ g(\omega) := \lambda\|\omega\|_1$. As per example 3.26 and §3.3.2 of Boyd & Vandenberghe,

   $$f^*(z) = (\|\cdot\|_1)^*(z) - \langle z, y\rangle = \begin{cases} -\langle z, y\rangle & \|z\|_\infty \le 1 \\ \infty & \text{otherwise} \end{cases} \qquad g^*(z) = \begin{cases} 0 & \|z\|_\infty \le \lambda \\ \infty & \text{otherwise} \end{cases}$$

Therefore,

$$D(z) = -f^*(-z) - g^*(X^T z) = \begin{cases} \langle z, y \rangle & \|z\|_\infty \leq 1 \text{ and } \|X^T z\|_\infty \leq 1 \\ \infty & \text{otherwise} \end{cases}$$

3. As suggested, define $f(v) := \sum_{i=1}^n \log(1+\exp(v_i))$ and $g(w) := \frac{\lambda}{2}\|w\|_2^2$. We already know the form for $g^*(z)$; it remains to compute $f^*(z)$.

$$f^*(z) = \sup_{v \in \mathbb{R}^n} \langle z, v \rangle - \sum_{i=1}^n \log(1+\exp(v_i))$$

Using this form as motivation, we note that $f^*(z) = \sum_{i=1}^n f^*(z_i)$ where

$$f^*(z_i) = \sup_{v_i \in \mathbb{R}^d} z_i v_i - \log(1+\exp(v_i))$$

Now

$$\nabla_{v_i}\big(z_i v_i - \log(1+\exp(v_i))\big) = z_i - \frac{\exp(v_i)}{1+\exp(v_i)} = z_i - 1 + \frac{1}{1+\exp(v_i)} = 0$$

Therefore,

$$v_i = \log(\frac{z_i}{1-z_i})$$

whence

$$f^*(z_i) = z_i \log(\frac{z_i}{1-z_i}) + \log(1-z_i) = \log({z_i}^{z_i}) - \log((1-z_i)^{z_i}) + \log(1-z_i)$$

Putting it all together,

$$f^*(z) = \sum_{i=1}^n f^*(z_i) = \sum_{i=1}^n \big[\log({z_i}^{z_i}) + \log((1-z_i)^{1-z_i})\big]$$

Thus, where $A := \text{diag}\,(y^i)X$, the Fenchel dual is given by

$$D(z) = -f^*(-z) - g^*(A^T z) = \sum_{i=1}^n \big[z_i \log(-z_i) - (1+z_i)\log(1+z_i)\big] - \frac{1}{2\lambda}\|A^T z\|_2^2$$

3.2 Stochastic Dual Coordinate Ascent

The dual of the SVM problem,

$$P(w) = \sum_{i=1}^N \max\{0, 1 - y^i w^T x^i\} + \frac{\lambda}{2}\|w\|^2,$$

is

$$D(z) = e^T z - \frac{1}{2\lambda} z^T Y X X^T Y z, \quad \text{s.t. } 0 \leq z_i \leq 1, \forall_i.$$

where e is a vector of ones, Y is diagonal matrix with the y^i values along the diagonal, and we have $w^* = \frac{1}{\lambda} X^T Y z^*$. Starting from *example_dual.m*, implement a dual coordinate optimization strategy to optimize the SVM objective. Hand in your code, report the optimal value of the primal and dual objectives with $\lambda = 1$, and report the number of support vectors

Hint: the objective function is a quadratic and the constraints are just lower and upper bounds. This lets you derive the optimal update for one variable with the other held fixed: solve for the value of z_i with a partial derivative of zero, and if this violates the constraints then the solution must be either $z_i = 0$ or $z_i = 1$ (depending on which one is lower).

3.3 Large-Scale Kernel Methods

The function *kernelRegression.m* implements kernel regression with the squared error, L2-regularizer, and Gaussian kernel. If you run your cross-validation code from Assignment 1 Question 1.2, you'll find that it achieves similar performance to using Gaussian RBFs.

1. Report the λ and σ reported using cross-validation on this previous assignment question. What are the (approximate) relationships between λ and σ between the two models (the one with Gaussian RBFs and the other with Gaussian kernels).
2. Implement the *subset of regressors* model for large-scale kernel methods we discussed in class. Hand in your code and report the qualitative performance (describe how well the model fits the data visually) for small and large values of the number of regressors m.
3. Implement the *random kitchen sink* model for large-scale kernel methods we discussed in class. Hand in your code and contrast the performance of this method with the subset of regressors model, for both large and small m.