CSCI 6968/4968: Homework 3

Assigned Tuesday October 17 2023. Due by 11:59pm Tuesday October 31 2023.

Create a Jupyter notebook for this assignment, and use Python 3. Write documented, readable and clear code (e.g. use reasonable variable names). Submit this notebook. You will be graded primarily based on the solutions and answers visible in the notebook, but the notebook must be runnable.

1. [70 points] You will compare, empirically, subgradient descent to stochastic subgradient descent, using the ℓ_2-regularized support vector machine optimization problem, which has objective function

$$f(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \big(1 - y_i(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)\big)_+ + \frac{\lambda}{2} \|\mathbf{w}\|_2^2.$$

   Here $\mathbf{x}_i \in \mathbb{R}^d$ are feature vectors and $y_i \in \{-1, 1\}$.

   Specifically, you will fit a SVM on the UCI Adult data set, where the task is to predict whether a person's income is above $50K$/yr based on census data, using both subgradient descent and stochastic subgradient descent, and compare the accuracy vs computational effort for the two methods. The computational effort will be measured by the number of data points touched in generating a particular set of model parameters.

   Answer the following problems *in your notebook* in full sentences and provide the plots asked for in the notebook.

   - What is the subdifferential of f, considered as a function of the vector $(\mathbf{w}, b)$? Give the complete subdifferential, not just a subgradient.
   - Implement a stochastic subgradient descent solver function `sgdmethod(X, y, subgradloss, subgradreg, regparam, w1, T, a, m)` that takes as input:
     - `X`, an array of size $n \times d$, each row of which is a feature vector,
     - `y`, an array with n rows, each row of which is the corresponding target,
     - `subgradloss(x, y, w)`, a function that computes a subgradient of the loss on a single training example at the current parameter vector,
     - `subgradreg(regparam, w)`, a function that computes a subgradient of the regularizer at the current parameter vector,
     - `regparam`, the regularization parameter λ,
     - `w1`, the initial guess for the parameter vector $(\mathbf{w}, b)$,
     - `T`, the number of iterations,
     - `a`, a parameter governing the step size as $\alpha_t = (1 + at)^{-1}$,
     - `m`, the minibatch size,

     and returns the sequence $\boldsymbol{\omega}_1 = (\mathbf{w}_1, b_1), \ldots, \boldsymbol{\omega}_T = (\mathbf{w}_T, b_T)$ of model parameters. Sample m training pairs without replacement to form each minibatch. Note that when $m = n_{\text{train}}$, `sgdmethod` actually implements the subgradient descent method.
   - Load the `a9a` training and testing data sets from `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html` using `sklearn.datasets.load_svmlight_file`. Remove the last column from the training data set, so both data sets have 122 features (the 123rd feature doesn't exist in the test data). Preprocess the features by scaling, as in previous homeworks, in order to increase the accuracy of the SVM model.
   - Take $\lambda = 1/n_{\text{train}}$ and, by looking at the convergence curves for different values of a on a subset of the training data, choose a parameter a that you think will perform well for the subgradient method on the entire training data set.

- Run the subgradient descent method using this value of a to 'convergence', as determined visually by the asymptoting out of $f(\boldsymbol{\omega}_t)$ to a flat line.
- Similarly choose an a appropriate for sgd with minibatch size of one (hereafter simply referred to as sgd) and run the method to convergence.
- Plot, on the same graph, the accuracy (percentage of correctly predicted labels) of:
  * The sequence of models returned by the subgradient descent method, on the training set.
  * The sequence of models obtained by averaging the last half of the models returned by the subgradient descent method up to iterate t, on the training set. That is, for each $t \geq 2$, plot the accuracy of the model with parameter $\frac{1}{t-\lfloor t/2 \rfloor+1} \sum_{i=\lfloor t/2 \rfloor}^{t} \boldsymbol{\omega}_i$. This is a more robust alternative to the model averaging we saw in class.
  * The sequence of models returned by the subgradient descent method, on the test set.
  * The sequence of models obtained by averaging the last half of the models returned by the subgradient method up to iterate t, on the test set.

  To facilitate comparison between the rates of convergence, let the x-axis be the number of datapoints used (counting multiple passes) to generate each model — e.g. the first model returned from the subgradient method touched n_{train} datapoints, the second touched $2n_{\text{train}}$ and so on. Label your data series and axes! You may find it useful to omit the first few points to more clearly compare the behaviors.
- Make a similar plot, but this time plot the performance of the sgd method on the training and test sets.
- Report the highest test accuracy for each of four methods (subgradient descent and stochastic subgradient descent with the last model returned, and with the averaged models), along with the number of datapoints used to obtain those accuracies.
- What observations do you have about the performance of sgd and the subgradient method, and the impact of averaging?[1]

2. [30 points] Consider the following iterative algorithm for minimizing a function f on a convex set C: given a current point $\mathbf{x}_t$, linearize the function f around $\mathbf{x_t}$, then minimize this linear approximation over C, using a quadratic regularization that forces us not to stray too far from $\mathbf{x}_t$. That is, given $\mathbf{g} \in \partial f(\mathbf{x}_t)$

$$\mathbf{x}_{t+1} = \text{argmin}_{\mathbf{u} \in C} f(\mathbf{x}_t) + \langle \mathbf{g}, \mathbf{u} - \mathbf{x}_t \rangle + \frac{1}{2\alpha_t} \|\mathbf{u} - \mathbf{x}_t\|_2^2.$$

   Explain the geometric intuition behind using this optimization problem to choose the update, and argue that this algorithm is exactly the projected subgradient method with stepsize given by α_t.

   Now consider a similar unconstrained optimization problem where we replace the proximal regularizer with a quadratic with Hessian $\mathbf{P}_t \succeq 0$ that may change from iteration to iteration,

$$\mathbf{x}_{t+1} = \text{argmin}_{\mathbf{u}} f(\mathbf{x}_t) + \langle \mathbf{g}, \mathbf{u} - \mathbf{x}_t \rangle + \frac{1}{2\alpha_t} \|\mathbf{P}_t^{1/2}(\mathbf{u} - \mathbf{x}_t)\|_2^2.$$

   Find an expression for the update $\mathbf{x}_{t+1}$. Explain the geometric intuition behind this choice of quadaratic regularization, and why this update scheme subsumes both Newton's algorithm[2] and AdaGrad.

[1] You may want to play around with ℓ_1 regularization and minibatch sizes to see how they affect performance as well.

[2] Specifically, the unguarded version of Newton's method where we pick a stepsize α_t rather than using exact line search as in the guarded/damped version.

3. (CSCI 6961 students) [50 points] We have mentioned practical problems with using Newton's method as an algorithm for solving general large smooth unconstrained convex optimization problems:

   - Storing $\nabla^2 f(\mathbf{x}) \in \mathbb{R}^{d\times d}$ is infeasible for large d: when $d = 50,000$ and the matrix is stored in single precision, this requires more than 9Gb.
   - Computing $\nabla^2 f(\mathbf{x})$ can be expensive: it could require making a pass over n data points, and even if it does not require touching the data, you could need to compute all d^2 entries individually.
   - Exactly solving linear systems is expensive: to invert a $d \times d$ matrix takes $\Omega(d^3)$ flops in practice.

   Luckily, in some applications, these problems can be ameliorated by using inexact solvers to determine approximate Newton's search directions. The idea is that $\mathbf{d}_t = \nabla^2 f(\mathbf{x}_t)^{-1}\nabla f(\mathbf{x}_t)$ is the minimizer of

   $$\min_{\mathbf{x}} \|\nabla^2 f(\mathbf{x}_t)\mathbf{x} - \nabla f(\mathbf{x}_t)\|_2^2. \tag{Newton}$$

   when $\nabla^2 f(\mathbf{x}_t)$ is invertible. We can therefore avoid the cost of a matrix inversion and instead use a gradient method to approximately solve (**??**). Since these algorithms only require the computation of matrix-vector products against $\nabla^2 f(\mathbf{x}_t)$, if the problem is structured nicely, we can avoid the need to even store $\nabla^2 f(\mathbf{x}_t)$.

   In practice, one uses the conjugate gradient descent (CG) algorithm to solve (**??**), because CG converges faster than steepest descent: the iterates $\mathbf{z}_s$ resulting from using CG to solve (**??**) satisfy[3]

   $$\|\nabla^2 f(\mathbf{x}_t)\mathbf{z}_T - \nabla f(\mathbf{x}_t)\|_2^2 \leq \left(1 - \frac{1}{\sqrt{\kappa}}\right)^{T-1} \|\nabla^2 f(\mathbf{x}_t)\mathbf{z}_1 - \nabla f(\mathbf{x}_t)\|_2^2.$$

   This method, called Newton-CG, is most useful when it uses less flops than the exact Newton's method to reach the same solution tolerance. Let us determine when we can expect it to be useful.

   Suppose that it suffices to approximately solve (**??**) to ε-suboptimality to see the same convergence rate to $f(\mathbf{x}_\star)$ using the exact Newton's method and Newton-CG. Show that if the gradients of f are bounded everywhere— $\|\nabla f(\mathbf{x})\|_2^2 \leq B$— then when the condition number of f satisfies

   $$\kappa = O\left(d^2 \ln\left(\frac{B}{\varepsilon}\right)^{-2}\right),$$

   Newton-CG started with $\mathbf{z}_1 = 0$ is preferable to the exact Newton's method. You will find the fact that $\ln(x) \leq x - 1$ when $x > 0$ to be useful.

[3] The dependence of the convergence rate on $\sqrt{\kappa}$ rather than κ is why CG converges faster than steepest descent– also, CG does not require line searches!