CPSC 540 Assignment 2 (due February 2)

Large-Scale Machine Learning

1. You can work in groups on 1-3 people on the assignments.
2. Place your names and student numbers on the first page.
3. Submit all answers as a single PDF file using the handin program (details on Piazza).
4. The PDF should answer the questions *in order*, and include relevant code in the appropriate place.
5. You will lose marks if answers are unclear or if the TA can't easily find the location of the answers.
6. Corrections/clarifications after the first version of this document is posted will be marked in red.
7. Acknowledge sources of help from outside of class/textbook (classmates, online material, papers, etc.).
8. All numbered questions are equally weighted.

Course Survey

Please fill out the survey located here:
`https://survey.ubc.ca/s/cs540/`

The purpose of this survey is to collect information to use for future offerings. It can be helpful when it comes time to argue for a larger room, that there should be an intermediate CPSC 440, that offering this course every year benefits many departments, etc.

1 Convex Functions and Norms

Most modern machine learning models are fit using some form of gradient method. Thus, it is important to study the problems of (i) how long will a gradient method take to converge and (ii) will a gradient method converge to a global optimum? *Norms* are an important tool to analyze the convergence rate of gradient descent while *convex functions* are useful to analyze whether a gradient method will find a global optimum.

The first part of this question asks you to prove several inequalities involving norms. The purpose of this question is to give you some practice manipulating these functions, as well as inequalities involving them, before we move on to proving convergence rates. If you are not familiar with norms, see the notes on norms on the course webpage. The second part of this question give you practice proving that functions are convex, which is important because convexity implies that gradient descent will converge to the optimal function value (this question is also a warm-up to the matrix/vector manipulations you will need to do later in the course for deep learning methods). The third part gives you practice proving convexity of some complicated-looking functions by writing them as a sequence of simple operations that preserve convexity.

1.1 Inequalities Involving Norms

Let x and y be length-d column-vectors. Show that the following inequalities hold:

1. Relationship between decreasing p-norms: $\|x\|_\infty \leq \|x\|_2 \leq \|x\|_1$
2. Relationship between increasing p-norms: $\|x\|_1 \leq \sqrt{d}\|x\|_2 \leq d\|x\|_\infty$
3. Triangle inequality for 2-norm: $\|x+y\|_2 \leq \|x\|_2 + \|y\|_2$
4. "Not the triangle inequality" inequality: $\frac{1}{2}\|x+y\|_2^2 \leq \|x\|_2^2 + \|y\|_2^2$

You should use the definitions of the norms, but should not use the triangle inequality or the known equivalences between these norms (since these are the things you are trying to prove). Hint: for many of these it's easier if you work with squared values (and you may need to "complete the square"). Beyond non-negativity of norms, it may also help to use the Cauchy-Schwartz inequality and to use that $\|x\|_1 = x^T \text{sign}(x)$.

1.2 Showing Convexity from Definitions

Without using the "operations that preserve convexity", show that the following functions are convex:

1. Quadratic	$f(x) = ax^2 + bx$	$x \in \mathbb{R}, a > 0$
2. Negative logarithm	$f(x) = -\log(ax)$	$x > 0$
3. Any norm	$f(x) = \|x\|_p$	$x \in \mathbb{R}^d$
4. Logistic regression	$f(w) = \sum_{i=1}^n \log(1 + \exp(-y_i w^T x_i))$	$w \in \mathbb{R}^d$
5. L2-regularized Poisson regression	$f(w) = \sum_{i=1}^n -\log p(y_i \| w, x_i) + \frac{\lambda}{2}\|w\|^2$	$w \in \mathbb{R}^d, \lambda \geq 0$

where in the last part y_i represents count data and $p(y_i|w, x_i) = \frac{\exp(y_i w^T x_i)\exp(-\exp(w^T x_i))}{y_i!}$.

Hint: Norms are not differentiable in general, so you cannot use the Hessian for the third one. For the last two, you can use the Hessian structure as we did for least squares in class.

1.3 Showing Convexity using Operations that Preserve Convexity

Use the results above and from class, along with the operations that preserve convexity, to show that the following functions are convex (with $\lambda \geq 0$):

1. $f(w) = \|Xw - y\|_p + \lambda\|w\|_q$ (regularized regression with arbitrary p-norm and q-norm)
2. $f(w) = \sum_{i=1}^N \max\{0, |w^T x_i - y_i| - \epsilon\} + \frac{\lambda}{2}\|w\|_2^2$ (support vector regression)
3. $f(x) = \max_{ijk}\{|x_i| + |x_j| + |x_k|\}$ (3 largest-magnitude elements)

2 Gradient Descent

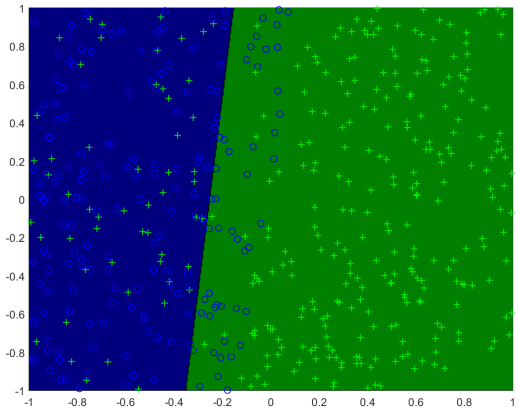
Gradient descent and its various generalizations are the main computational tools used to fit parameters in machine learning. One advantage of gradient-based strategies is that they are able to solve problems that do not have closed-form solutions. A second advantage of these methods is that they scale elegantly in the dimensionality d of problem. For example, while forming $X^T X$ in a least squares problem costs $O(nd^2)$, computing the gradient only costs $O(nd)$. Since the size of the dataset is $O(nd)$, we (basically) can apply gradient descent to any dataset we can store. In addition to working even if d is huge, it's possible to show that not too many iterations of gradient descent are required to minimize sufficiently-regularized objectives.

The first part of the question gets you to explore the effect of the loss function for fitting a linear model to a binary dataset, where as with robust regression we again see that the 'default' choice is sub-optimal. The second part of this question gives you practice proving convergence properties of gradient descent methods.

Unfortunately, despite its appealing theoretical properties, in practice a naive implementation of gradient descent will often have frustratingly-bad performance. The third part of the question considers more difficult tasks, and asks you to modify the gradient descent code itself to improve its performance.

2.1 Loss Functions

Download and exapnd *a2.zip*, then run *example_extreme* in Matlab. This loads a binary classification dataset with 2 features, then fits a leasts squares model (as in A1) and then a logistic regression model (with gradient descent). It reports the training error of these models and produces plots similar to the one below:



This is a scatterplot of the feature values across all the training examples. The x-axis is given by the first feature, the y-axis is given by the second feature, and the symbol ('+' or 'o') gives the class label. The green region shows the area of the space that the model labels '+', and the blue region shows the area of the space that the model labels 'o'. In this plot, we see that in half the space we only have '+' values while in the other half of the space we have mostly 'o' values. In this question we'll focus on the training set error, since with several hundred datapoints and a low-dimensional linear model we are very unlikely to overfit much.

The usual input to a gradient method is a function that, given w, returns $f(w)$ and $\nabla f(w)$. See *logisticGrad* in the *logisticRegression* function for an example. Note that the *logisticRegression* function also calls the function *derivativeCheck* which numerically check the accuracy of the gradient for a particular w, since implementing gradients is often error-prone.

Normally, logistic regression outperforms the squared error on binary classification problems, but for this dataset logistic regression actually performs worse. This is because logistic regression treats the classes symmetrically, but in this case the 'o' class is contaminated by the '+' class. A common *asymmetric* loss is the extreme-value loss. Instead of assuming that $p(y_i|w, x_i) = 1/(1 + \exp(-w^T x_i))$, this function assumes that

$$p(y_i = 1|w, x_i) = 1 - \exp(-\exp(w^T x_i)).$$

1. Derive the (asymmetric) loss function (i.e., negative log-likelihood) that corresponds to this assumption, and make a new model (*extremeLoss*) that uses this loss. Hand in the code, and report the training error with this loss function.

2. While the extreme loss improves performance on this dataset, it does not find the optimal linear model. Design a new (differentiable) loss function that leads to a linear model with a lower training error on this dataset. Hand in the code, and report the training error with this loss function.

Hint: it's possible to get an error under 14%.

2.2 Rate of Convergence

In class we showed that gradient descent achieves a linear convergence rate if the gradient is Lipschitz continuous and the function is strongly-convex. This question first asks you to prove a linear convergence rate under weaker assumptions. It then turns to the question of whether the parameters x^t converge under the usual strong-convexity assumptions, and fninally asks you to prove a faster rate in terms of the parameters. For a list of properties of convex functions and strongly-convex functions, as well as functions with Lipschitz-continuous gradients, see the convexity notes on the course webpage.

1. We'll say that a function is *strongly-invex* if for all x and some $\mu > 0$ it holds that

$$\frac{1}{2}\|\nabla f(x)\|^2 \geq \mu(f(x) - f^*),$$

where f^* is the function value achieved by any optimal solution x^*. For differentiable functions, invex functions are the set of functions where all stationary points are global minimizers (i.e., all convex functions are invex but there are invex functions that are not convex). If a function is μ-strongly convex then it is also μ-strongly invex, but some μ-strongly invex functions are not strongly-convex at all. For example, least squares when X^TX is *not* invertible is convex and also μ-strongly invex but is *not* strongly-convex. Also, note that μ-strongly invex function do not need to be convex.

Assume that a function f has an L-Lipschitz continuous gradient, has at least one solution x^*, and is μ-strongly invex. If we use gradient descent with a constant step-size of $1/L$,

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

show that is has the global linear convergencer rate

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

2. Instead of the optimal function value f^*, we are often interested in finding the optimal parameter vector x^*. For example, we might want to show how quickly $\|x^k - x^*\|$ converges to zero. This is often more challenging in general, but for strongly-convex functions we can convert a rate in terms of function values into a rate in terms of the parameter vector. In particular, consider an algorithm that achieves a linear convergence rate in terms of function values of

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

Show that for strongly-convex functions this implies a linear convergence rate in terms of the parameter vector of

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

3. The rate above will often be slower than the rate we can show if we are able to directly analyze $\|x^t - x^*\|^2$. For example, consider a μ-strongly convex function with an L-Lipschitz continuous gradient and the gradient descent iteration

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

If we used the argument above and the gradient descent proof from class, we would get a rate of

$$\|x^t - x^*\| = O\left(\left(1 - \frac{\mu}{L})\right)^{t/2}\right).$$

Show that it's possible to get a faster rate of

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

Hint: expand the squared norm like we did in class. You will have to use an inequality from the notes which simultaneously uses the Lipschitz and strong-convexity assumptions.

2.3 Implementation and Improving Performance

The function *example_gradient* loads a standard ML benchmark binary classification, and tries to fit an ℓ_2-regularized logistic regression model to it. Unfortunately, it doesn't work at all because the *gradDesc* method fails catastophically: not only is the norm of the gradient far away from zero, but the final value it finds is *worse* than the initial value with $w = 0$.

Modify the function *gradDesc* so that it is able to find a parameter vector w that satisfies $\|\nabla f(w^t)\|_\infty \leq 0.1$ with less than 100 evaluations of the function, while maintaining that the cost per function evaluation is still $O(d)$ in terms of d..

Hint: you will probably need to find some reasonable way to set the step-size (adaptive, Armijo, etc.) and you will probably also need to use one of the tricks we discussed in class to improve performance (Nesterov's algorithm or practical approximations to Newton's method). Note that not all of these tricks work equally well. If are unsure where to start, you may want to look ahead to the code provided in Question 3.3.

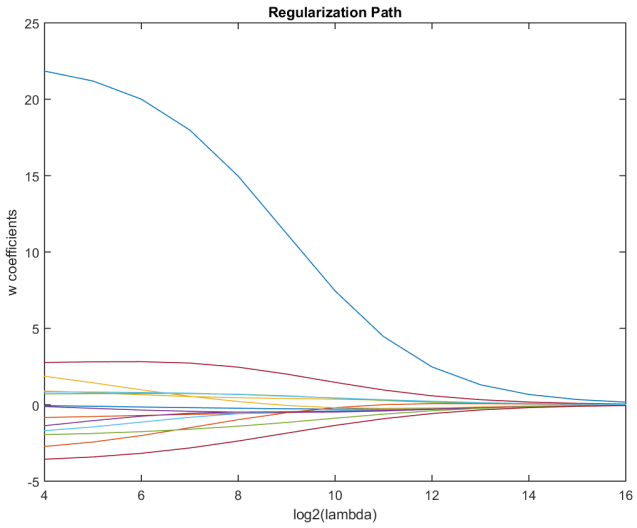
3 Structured Sparsity and Non-Smooth Optimization

As discussed in class, L1-regularization leads to *sparsity* in the parmaeter vector, which can be used to prune irrelevant features. *Structured sparsity* generalizes L1-regularization to allow more general patterns of sparsity. However, we can't apply gradient descent to these objectives because they are non-smooth. In this question you will explore two sparsity patterns as well as some common algorithms for fitting models with sparsity-inducing regularizers.

First, you will make a simple implementation of the coordinate descent method for L1-regularization. Second, you will analyze the convergence rate of block coordinate descent under uniform and non-uniform sampling. Finally, you will explore using *group* L1-regularization to encourage sparsity in groups of variables.

3.1 L1-Regularization and Proximal-Gradient Methods

The function *example_lasso* loads a classic regression problem and fits an L2-regularized least squares estimator to it for various values of λ, then plots the "regularization path" of the coefficients against the regularization parameter.



If you zoom in on the largest regularization parameter, you will see a weird effect of L2-regularization: instead of setting the values of the coefficients to zero (as λ gets large) it tries to set them so that they have the *same magnitudes*.

In class we discussed the alternate L1-regularized estimator,

$$\underset{w\in\mathbb{R}^d}{\operatorname{argmin}} \frac{1}{2}\|Xw - y\|^2 + \lambda\|w\|_1,$$

which sets variables w_j to exactly zero but is non-smooth. While we could formulate this problem as a quadratic program, to solve instances of this problem where d is very large we want to take advantage of the simplicity of the L1-norm.

1. One way to do this is with proximal-gradient methods. The function *proxGradLassso* implements a proximal-gradient method (using a step-size of $1/L$) for this problem. Modify the demo to use *proxGradLasso* to compute the L1-regularization path instead of the L2-regularization path. Hand in the updated plot.

2. Unfortunately, even for this simple problem the proximal-gradient method with a step-size of $1/L$ is not very effective for small values of λ (it takes many iterations to reach an accurate solution). Many of the tricks from Question 2.3 can be adapted to improve its performance, but another strategy is to notice that the problem has the form

$$\underset{w\in\mathbb{R}^d}{\operatorname{argmin}} f(Xw - y) + \sum_{j=1}^{d} g_j(x_j),$$

for a smooth function f and a set of non-smooth functions g_j. This allows us to apply *coordinate optimization* methods. Write a function, *coordOptLasso* that implements a coordinate optimization method for this problem (using random selection of the coordinate to update). Make sure that the iteration cost is $O(n)$ so that the iterations are d times faster than the proximal-gradient iteration cost of $O(nd)$. Hand in your function.

3.2 Converge Rate of Block Coordinate Descent

Consider the optimization problem

$$\underset{w\in\mathbb{R}^d}{\operatorname{argmin}} f(w),$$

where f is twice-differentiable. We'll partition our variables into k disjoint 'blocks'

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix},$$

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

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

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

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

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

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

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 step-size of $1/L_{b_t}$ then we obtain a faster convergence rate provided that some $L_b \neq L$.

3.3 Group L1-Regularization

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 0.28 (depending on your implementation of *gradDesc*).

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 λ set to a power of 10. Also report the number of non-zero parameters in the model and the number of original features that the model uses for the best value of λ.

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 for the best value of λ.

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 best validation error achievable with λ set to a power of 10. Also report the number of non-zero parameters in the model and the number of original features that the model uses for the best value of λ.

Hint: the soft-threshold operator with a threshold of τ for a group g is given by

$$\frac{w_g}{\|w_g\|}\max\{0, \|w_g\| - \tau\}.$$

4 Stochastic Subgradient Methods

Stochastic gradient methods are an appealing training strategy when the number of training examples n is very large. They have appealing theoretical properties in terms of the training and testing objective, even when n is infinite. However, they can prove difficult to get working in practice, and are much reliable than the methods above.

In this quesiton, you'll explore how different techniques affect the performance of stochastic gradient method. You'll explore using different step size selection schemes and using the average of pervious iterations. Finally, you'll you'll prove a convergence rate for a variant of the method for strongly-convex objectives.

4.1 Averaging and Step-Size Strategies

Consider minimizing the SVM objective function,

$$\min_w F(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n}\max\{0, 1 - y_i w^T x_i\}.$$

This objective function is λ-strongly convex but non-smooth so it is a good candidate for stochastic subgradient methods. The function *example_svm* tries to minimize this function (on a problem arising from quantum physics) using a stochastic subgradient method, using a step-size of $\frac{1}{\lambda t}$, and plots the objective function against the number of "passes" through the data as it goes. Notice that it performs terribly: it actually *increases* the objective function quite substantially on the first iteration and typically never gets down to its initial value. In this question we'll explore methods for improving the performance.

1. In our analysis of the stochastic subgradient method, we proved a convergence result for the *average* of the w^t variables rather than the final value. Make a new function, *svmAvg*, that reports the performance based on the running average of the w^t values rather than the current value. Hand in the modified part of the code and a plot of the performance with averaging.

2. While averaging all the iterations smoothes the performance, the final performance isn't too much better. This is because it places just as much weight on the early iterations (w^0, w^1, and so on) as it does on the later iterations. A common variant is to exclude the early iterations from the average. For example, we can start averaging once we have get half way to *maxIter*. Modify your *svmAvg* code to do this, and hand in the modified parts of the code and a plot of the performance this "second-half" averaging.

3. The modification above allows you to find a better solution than the initial w^0, and it achieves the theoretically-optimal rate in terms of the accuracy ϵ, but the practical performance is still clearly bad. Modify the *svm* function to try to optimize its performance. hand in your new code, a description of the modifications you found most effective, and a plot of the performance with your modifications.

4.2 Efficient Projected Stochastic Subgradient with Sparse Gradients

Consider a variant of the SVM problem where the x_i are *sparse*. In class, we showed how to efficiently apply stochastic subgradient methods in this setting by using the representation $w^t = \beta^t v^t$. Now consider a case where we know an L2 ball that contains the optimal solution. In other words, we know a τ such that $\|w^*\| \leq \tau$. If τ is small enough, we can use a *projected* stochastic subgradient method (projecting onto the L2-ball):

$$w^{t+\frac{1}{2}} = w^t - \alpha_t g^t - \alpha_t \lambda w^t$$

$$w^{t+1} = \begin{cases} w^{t+\frac{1}{2}} & \text{if } \|w^{t+\frac{1}{2}}\| \leq \tau \\ \frac{\tau}{\|w^{t+\frac{1}{2}}\|} w^{t+\frac{1}{2}} & \text{if } \|w^{t+\frac{1}{2}}\| > \tau \end{cases}$$

By constraining the w^t to a smaller set, this can sometimes dramatically improve the performance in the early iterations. However, the projection operator is a full-vector operation so this would substantially slow down the runtime of the method. Derive a recursion that implements this algorithm without using full-vector operations.

Hint: you may need to track more information than β^t and v^t.