CPSC 540 Assignment 2

The assignment instructions are the same as for the previous assignment, but for this assignment you can work in groups of 1-3. However, please only hand in one assignment for the group.

1. Name(s): Cody Griffith, Tim Jaschek, Ziming Yin
2. Student ID(s): 88416169, 91220160, 88489166

1 Calculation Questions

1.1 Convexity

Show that the following functions are convex, by only using one of the definitions of convexity (i.e., without using the "operations that preserve convexity" or using convexity results stated in class):[1]

1. Upper bound on second-order Taylor expansion: $f(v) = f(w) + \nabla f(w)^T(v-w) + \frac{L}{2}\|v-w\|^2$.
   To avoid ambiguity in the notation, let us define $\varphi(v) := f(w) + \nabla f(w)^T(v-w) + \frac{L}{2}\|v-w\|^2$. This is necessary as f itself does not need to be convex. However, we will show that a second order Taylor expansion has to be convex. Since we are using a second order Taylor expansion, we have that $f \in C^2$ and so is φ. Therefore gradient and Hessian are well defined and we compute

   $$\nabla\varphi(v) = \nabla f(w) + L(v-w)$$
   $$\nabla^2\varphi(v) = LI \succeq 0,$$

   which gives us that φ is convex for $L > 0$.

2. Probit regression: $f(w) = \sum_{i=1}^n -\log p(y_i|w, x_i)$ (where the probability is defined as in the last assignment).
   From homework 1, we have,

   $$\nabla^2 f(w) = X^T D X, \quad D_{ij} = \delta_{i,j}\left(-\frac{p_i'}{c_i} + \frac{p_i^2}{c_i^2}\right).$$

   Here we have that $p_i' = -y^i w^T x^i p_i$ and that $D_{ii} > 0$. Thus,

   $$v^T\nabla^2 f(w)v = \left(D^{1/2}Xv\right)^T D^{1/2}Xv = \|D^{1/2}Xv\|_2^2 \geq 0.$$

   This shows that f is convex.

[1] That C^0 convex functions are below their chords, that C^1 convex functions are above their tangents, or that C^2 convex functions have a positive semidefinite Hessian.

3. Maximum function: $f(w) = \max_{j\in\{1,2,\dots,d\}} w_j$.
   Here we only know that $f \in C^0$ so pick any w, v and $\theta \in [0,1]$,

$$\begin{aligned} f(\theta w + (1-\theta)v) &= \max_{j=1,\dots,d}\{\theta w_j + (1-\theta)v_j\}, \\ &\leq \max_{j=1,\dots,d}\{\theta w_j\} + \max_{j=1,\dots,d}\{(1-\theta)v_j\}, \\ &= \theta f(w) + (1-\theta) f(v). \end{aligned}$$

   Which gives us that f is convex.

Hint: Max are is not differentiable in general, so you cannot use the Hessian for the last one. Hint: Max are is not differentiable in general, so you cannot use the Hessian for the last one. For the second one, you can assume that $PDF(z) \geq -z \cdot \text{CDF}(z)$ (where PDF(z) is the PDF of a standard normal and CDF(z) is the CDF).

Show that the following functions are convex (you can use results from class and operations that preserve convexity if they help):

4. 1-class SVMs: $f(w, w_0) = \sum_{i=1}^N [\max\{0, w_0 - w^T x_i\} - w_0] + \frac{\lambda}{2}\|w\|_2^2$, where $\lambda \geq 0$ and w_0 is a variable.
   From class we know that,
   - Max is convex over a convex function: $\max\{0, w_0 - w^T x_i\}$.
   - Affine transformations of convex functions are convex: $\max\{0, w_0 - w^T x_i\} - w_0$.
   - Sums of convex functions are convex: $\sum_{i=1}^N [\max\{0, w_0 - w^T x_i\} - w_0]$.
   - Norm-squared is convex: $\frac{\lambda}{2}\|w\|_2^2$.
   - Sums of convex are convex: $f(w, w_0) = \sum_{i=1}^N [\max\{0, w_0 - w^T x_i\} - w_0] + \frac{\lambda}{2}\|w\|_2^2$.

5. Mixed-norm regularization: $f(w) = \|w\|_{p,q} = \left(\sum_{g\in\mathcal{G}} \|w_g\|_q^p\right)^{\frac{1}{p}}$.
   For any $\theta \in [0,1]$ and $v, w \in \mathbb{R}^d$ it holds

$$\begin{aligned} f(\theta w + (1-\theta)v) &= \left(\sum_{g\in\mathcal{G}} \|\theta w_g + (1-\theta)v_g\|_q^p\right)^{\frac{1}{p}} \\ &\leqslant \left(\sum_{g\in\mathcal{G}} (\theta\|w_g\|_q + (1-\theta)\|v_g\|_q)^p\right)^{\frac{1}{p}} \\ &\leqslant \left(\sum_{g\in\mathcal{G}} (\theta\|w_g\|_q)^p\right)^{\frac{1}{p}} + \left(\sum_{g\in\mathcal{G}} ((1-\theta)\|v_g\|_q)^p\right)^{\frac{1}{p}} \\ &= \theta\left(\sum_{g\in\mathcal{G}} \|w_g\|_q^p\right)^{\frac{1}{p}} + (1-\theta)\left(\sum_{g\in\mathcal{G}} \|v_g\|_q^p\right)^{\frac{1}{p}} \\ &= \theta f(w) + (1-\theta) f(v). \end{aligned}$$

   The firsy inequality followed by the triangle inequality for the $\ell^q(\mathbb{R}^d)$-norm and the second inequality because of the triangle inequality for the $\ell^p(\mathbb{R}^{|\mathcal{G}|})$-norm.

6. Minimum entropy over pairs of variables: $f(w) = \max_{\{i,j \mid i \neq j\}}\{w_i \log w_i + w_j \log w_j\}$ subject to $w_i > 0$ for all i.
Here we note that $w_i \log w_j$ is convex from,

$$\nabla^2(w \log(w)) = \frac{1}{w} > 0.$$

So then from class we have,

- sums of convex functions are convex:$w_i \log w_i + w_j \log w_j$.
- Max is convex over a convex function: $f(w) = \max_{\{i,j \mid i \neq j\}}\{w_i \log w_i + w_j \log w_j\}$.

1.2 Convergence of Gradient Descent

For these questions it will be helpful to use the "convexity inequalities" notes posted on the webpage.

1. In class we showed that if ∇f is L-Lipschitz continuous and f is strongly-convex, then with a step-size of $\alpha_k = 1/L$ gradient descent has a convergence rate of

$$f(w^k) - f(w^*) = O(\rho^k).$$

Show that under these assumptions that a convergence rate of $O(\rho^k)$ in terms of the function values implies that the iterations have a convergence rate of

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

Recall the two inequalities:

$$\text{Progress bound: } f(w^{k+1}) \leq f(w^k) - \frac{1}{2L}\|\nabla f(w^k)\|^2$$
$$\mu\text{-convexity: } \|x - y\| \leq \frac{1}{\mu}\|\nabla f(x) - \nabla f(y)\|$$

Note that $\nabla f(w^*) = 0$ and therefore applying the two inequalities one after another, we find,

$$\|w^k - w^*\|^2 \leq \frac{1}{\mu^2}\|\nabla f(w^k) - \nabla f(w^*)\|^2 \leq \frac{2L}{\mu^2}\left[f(w^k) - f(w^{k+1})\right].$$

Thus,

$$\begin{aligned}\|w^k - w^*\| \leq& \frac{\sqrt{2L}}{\mu}\sqrt{f(w^k) - f(w^{k+1})},\\ \leq& \frac{\sqrt{2L}}{\mu}\sqrt{f(w^k) - f(w^*) - (f(w^{k+1}) - f(w^*))},\\ \leq& \frac{\sqrt{2L}}{\mu}\sqrt{f(w^k) - f(w^*)},\end{aligned}$$

where the last inequality follows as $f(w^{k+1}) - f(w^*) \geqslant 0$. Finally, using that $f(w^k) - f(w^*) \sim O(\rho^k)$, we can conclude that $\|w^k - w^*\| \sim O(\rho^{k/2})$.

2. A basic variation on the Armijo line-search is to set the step-size α_k to be $\alpha_k = (0.5)^p$, where p is the smallest constant such that

$$f(w^k - \alpha_k \nabla f(w^k)) \leq f(w^k) - \frac{\alpha_k}{2}\|\nabla f(w^k)\|^2.$$

Show that if ∇f is Lipschitz-continuous and f is bounded below, that setting α_k in this way means gradient descent requires at most $t = O(1/\epsilon)$ iterations to find at least one iteration k with $\|\nabla f(w^k)\|^2 \leq \epsilon$. Hint: you'll first want to figure out a progress bound of the form

$$f(w^{k+1}) \leq f(w^k) - \gamma \|\nabla f(w^k)\|^2,$$

for some constant γ that holds for all iterations. It may also help to recognize that if a function is L-Lipschitz continuous that it is also L'-Lipschitz continuous for any $L' \geq L$.

Answer: Since ∇f is lipschitz continuous, we can apply the descent lemma on w^k and w^{k+1}

$$\begin{aligned} f(w^{k+1}) &\leqslant f(w^k) + \nabla f(w^k)^T (w^{k+1} - w^k) - \frac{L}{2}\|w^{k+1} - w^k\|^2 \\ &\leqslant f(w^k) - \alpha_k \nabla f(w^k)^T \nabla f(w^k) - \frac{L\alpha_k^2}{2}\|\nabla f(w^k)\|^2 \\ &= f(w^k) + \left(\frac{L\alpha_k^2}{2} - \alpha_k\right)\|\nabla f(w^k)\|^2 \end{aligned}$$

Recap that we set the step-size α_k to be $\alpha_k = (0.5)^p$, where p is the smallest constant such that

$$f(w^k - \alpha_k \nabla f(w^k)) \leq f(w^k) - \frac{\alpha_k}{2}\|\nabla f(w^k)\|^2.$$

Therefore, by minimality in the choice of α_k, we must have

$$\frac{\alpha_k}{2} \geqslant -\left(\frac{L\alpha_k^2}{2} - \alpha_k\right) \Rightarrow \alpha_k \geqslant \frac{1}{L}.$$

where we divided by α_k which is fine since it cannot be zero. This bound keeps α_k from being too small. Now set $\gamma := \min_{k=0,\ldots,t} \frac{\alpha_k}{2} \geqslant (2L)^{-1}$. Then

$$f(w^{k+1}) \leqslant f(w^k) - \gamma \|\nabla f(w^k)\|^2.$$

We use here that since f is bounded below, there exists some $f^* \leq f(w)$ for all w and a telescoping sum argument yields

$$\min_{k=0,\ldots,t} \|\nabla f(w^k)\| \leqslant \frac{1}{t\gamma}\left(f(w^0) - f(w^t)\right) \leqslant \frac{1}{t\gamma}\left(f(w^0) - f(w^*)\right) = C\frac{1}{t}.$$

Thus $\min_{k=0,\ldots,t} \|\nabla f(w^k)\| \sim O(1/t)$ which implies that gradient descent requires at most $t = O(1/\epsilon)$ iterations to find at least one iteration k with $\|\nabla f(w^k)\|^2 \leq \epsilon$

3. Show that if ∇f is Lipschitz-continuous and f is convex (but not necessarily strongly-convex or PL), that if we run gradient descent for k iterations with $\alpha_k = 1/L$ we have

$$f(w^k) - f(w^*) = O(1/k).$$

Hint: you will have to use one the C^1 definition of convexity in our usual progress bound coming from Lipschitz-continuity of the gradient. You can try to turn this into a telescoping sum in terms of $\|w^k - w^*\|^2$ (which invovles "completing the square"). Finally, note that our usual progress bound also gives that $f(w^{k+1}) \leq f(w^k)$ for all k.

Answer: By convexity of f we must have $f(w^*) \geqslant f(w^k) + \nabla f(w^k)^T (w^* - w^k)$. Substituting this in our progress bound that we obtained from Lipschitz continuity of the gradient yields

$$\begin{aligned} f(w^{k+1}) &\leqslant f(w^k) - \frac{1}{2L}\|\nabla f(w^k)\|^2 \\ &\leqslant f(w^*) + \nabla f(w^k)^T (w^k - w^*) - \frac{1}{2L}\|\nabla f(w^k)\|^2 \end{aligned}$$

Following the hint, we want to complete the square on the right hand side, which involves a few tricks:

$$\begin{aligned}
f(w^{k+1}) &\leqslant f(w^*) + \nabla f(w^k)^T(w^k - w^*) - \frac{1}{2L}\|\nabla f(w^k)\|^2 \\
&= f(w^*) + L(\frac{1}{L}\nabla f(w^k))^T(w^k - w^*) - \frac{L^2}{2L}\|\frac{1}{L}\nabla f(w^k)\|^2 \\
&= f(w^*) + \frac{L}{2}\left(\frac{1}{2}(\frac{1}{L}\nabla f(w^k))^T(w^k - w^*) - \|\frac{1}{L}\nabla f(w^k)\|^2 + \|w^k - w^*\|^2 - \|w^k - w^*\|^2\right) \\
&= f(w^*) + \frac{L}{2}\left(-\|\frac{1}{L}\nabla f(w^k) - (w^k - w^*)\|^2 + \|w^k - w^*\|^2\right) \\
&= f(w^*) + \frac{L}{2}\left(-\| - w^{k+1} + w^*\|^2 + \|w^k - w^*\|^2\right) \\
&= f(w^*) + \frac{L}{2}\left(\|w^k - w^*\|^2 - \|w^{k+1} - w^*\|^2\right)
\end{aligned}$$

After this nasty computation we are almost done. summing the above expression for $j = 1, \ldots, k$ gives

$$\sum_{j=1}^{k} f(w^j) - f(w^*) \leqslant \frac{L}{2}\left(\|w^0 - w^*\|^2 - \|w^k - w^*\|^2\right) \leqslant \frac{L}{2}\|w^0 - w^*\|.$$

Note that the upper bound does not depend on k. Furthermore, each term of the sum on the left hand side is larger or equal than $f(w^k) - f(w^*)$, simply because the step size $\alpha_k = 1/L$ justifies that $f(w^j)$ is decreasing in k. Therefore

$$k\left(f(w^k) - f(w^*)\right) \leqslant \sum_{j=1}^{k} f(w^j) - f(w^*) \leqslant \frac{L}{2}\|w^0 - w^*\|.$$

Dividing both sides by k yields the claim that $f(w^k) - f(w^*) \sim O(1/k)$.

1.3 Beyond Gradient Descent

1. We can write the proximal-gradient update as

$$\begin{aligned}
w^{k+\frac{1}{2}} &= w^k - \alpha_k \nabla f(w^k) \\
w^{k+1} &= \operatorname*{argmin}_{v \in \mathbb{R}^d} \left\{ \frac{1}{2}\|v - w^{k+\frac{1}{2}}\|^2 + \alpha_k r(v) \right\}.
\end{aligned}$$

Show that this is equivalent to setting

$$w^{k+1} \in \operatorname*{argmin}_{v \in \mathbb{R}^d} \left\{ f(w^k) + \nabla f(w^k)^T(v - w^k) + \frac{1}{2\alpha_k}\|v - w^k\|^2 + r(v) \right\}.$$

Answer: Plugging the definition of $w^{k+\frac{1}{2}}$ into the one of w^k we obtain

$$w^{k+1} \in \operatorname*{argmin}_{v \in \mathbb{R}^d} \left\{ \frac{1}{2}\|v - w^k + \alpha_k \nabla f(w^k)\|^2 + \alpha_k r(v) \right\} \tag{1}$$

Using that $\|x-y\|_2^2 = \sum_{i=1}^d (x_i - y_i)^2 = \sum_{i=1}^d (x_i^2 - 2x_i y_i + y_1^2) = \|x\|_2^2 + \|y\|_2^2 - 2\langle x, y\rangle$ for all $x, y \in \mathbb{R}^d$, we conclude that equation (1) is equivalent to

$$\begin{aligned} w^{k+1} &\in \operatorname*{argmin}_{v\in\mathbb{R}^d} \left\{ \frac{\alpha_k^2}{2}\|\nabla f(w^k)\| + \alpha_k \nabla f(w^k)^T(v - w^k) + \frac{1}{2}\|v - w^k\| + \alpha_k r(v) \right\} \\ &= \operatorname*{argmin}_{v\in\mathbb{R}^d} \left\{ \frac{\alpha_k}{2}\|\nabla f(w^k)\| + \nabla f(w^k)^T(v - w^k) + \frac{1}{2\alpha_k}\|v - w^k\| + r(v) \right\} \\ &= \operatorname*{argmin}_{v\in\mathbb{R}^d} \left\{ f(w^k) + \nabla f(w^k)^T(v - w^k) + \frac{1}{2\alpha_k}\|v - w^k\|^2 + r(v) \right\}, \end{aligned}$$

where we obtained the first equality by dividing the terms in the argmin by α_k and the second equality by adding the constant $f(w^k) - \frac{\alpha_k}{2}\|\nabla f(w^k)\|^2$. Both of these operations do not effect the argmin.

2. In class we showed that if ∇f is coordinate-wise L-Lipschitz and satisfies PL then randomized coordinate optimization with a step-size of $1/L$ satisfies

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

Consider a C^2 function satisfying PL where coordinate i has its own Lipschitz constant L_j,

$$\nabla^2_{jj} f(w) \leq L_j.$$

Consider using a step-size of $1/L_{j_k}$ and sampling j_k proportional to the L_j, $p(j_k = j) = \frac{L_j}{\sum_{j'} L_{j'}}$. Show that this gives a faster convergence rate. Hint: the previous result corresponds to using $L = \max_j\{L_j\}$.

Answer: First we establish a coordinate-wise descent lemma (holds true because of the coordinate-wise L-Lipschitz assumption):

$$f(w^{k+1}) \leq f(w^k) + \nabla_j f(w^k)(w^{k+1} - w^k)_j + \frac{L_j}{2}(w^{k+1} - w^k)_j^2.$$

But here we notice that using a coordinate-wise gradient descent with $\alpha_k = 1/L_j$ we have,

$$(w^{k+1} - w^k)_j = -\frac{1}{L_j}|\nabla_j f(w^k)|^2.$$

Applying this to the descent lemma gives us a progress bound for each coordinate,

$$f(w^{k+1}) \leq f(w^k) - \frac{1}{2L_j}|\nabla_j f(w^k)|^2.$$

Let us now assume that we sample the coordinate that we want to optimize on with respect to the density function $p(j_k = j) = L_j / \sum_{m=1}^d L_m$. The previous inequality then looks like

$$f(w^{k+1}) \leq f(w^k) - \frac{1}{2L_{j_k}}|\nabla_{j_k} f(w^k)|^2,$$

where we now take the expectation with respect to the above density function.

$$\mathrm{E}[f(w^{k+1})] = E[f(w^k)] - \sum_{j=1}^d p(j_k = j)\frac{1}{2L_j}|\nabla_j f(w^k)|^2 \leq f(w^k) - \frac{1}{2}\sum_{j=1}^d \frac{1}{\sum_m L_m}|\nabla_j f(w^k)|^2.$$

Here we choose to write $\overline{L} = \frac{1}{d}\sum_m L_m$ and thus,

$$\mathrm{E}[(f(w^{k+1})] \leq f(w^k) - \frac{1}{2d\overline{L}}\|\nabla f(w^k)\|^2.$$

Lastly, using the PL inequality we have both the following inequalities since $f(w^*)$ exists,

$$\mathrm{E}[f(w^{k+1})] - f(w^*) \leq \left(1 - \frac{\mu}{d\overline{L}}\right)(f(w^k) - f(w^*)),$$
$$\mathrm{E}[f(w^{k})] - f(w^*) \leq \left(1 - \frac{\mu}{d\overline{L}}\right)^k (f(w^0) - f(w^*)).$$

Which now we have that the coordinate-wise gradient descent has convergence rate,

$$O\left(\frac{d\overline{L}}{\mu}\log(1/\epsilon)\right).$$

This is strictly faster as long as $\overline{L} < L$ and this will happen as long as at least one $L_j \neq L_i$ for some i but is exactly the same if all $L_j = L$.

3. Consider an SVM problem where the x^i are *sparse*. In the bonus slides, we show how to efficiently apply stochastic subgradient methods in this setting by using the representation $w^k = \beta^k v^k$. 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):

$$w^{k+\frac{1}{2}} = w^k - \alpha_k g^k - \alpha_k \lambda w^k$$
$$w^{k+1} = \begin{cases} w^{k+\frac{1}{2}} & \text{if } \|w^{k+\frac{1}{2}}\| \leq \tau \\ \frac{\tau}{\|w^{k+\frac{1}{2}}\|} w^{k+\frac{1}{2}} & \text{if } \|w^{k+\frac{1}{2}}\| > \tau \end{cases}$$

By constraining the w^k to a smaller set, this can sometimes dramatically improve the performance in the early iterations.[2] 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^k and v^k.

Answer: To improve, we should track $w^k = \beta^k \nu^k$ and $\gamma^k = \|w^k\|_2^2$. These quantities each are: β^k-scaling, ν^k-updating vector and γ^k-magnitude. This rule should be,

$$\beta^{k+1/2} = (1 - \alpha_k\lambda)\beta^k, \quad \gamma^{k+1/2} = \|w^{k+1/2}\|_2^2, \quad \nu^{k+1/2} = \nu^k.$$

But notice that,

$$\begin{aligned}\gamma^{k+1/2} =& \|(1-\alpha_k\lambda)w^k - \alpha_k g^k\|_2^2, \\ =& (1-\alpha_k\lambda)\|w^k\|_2^2 - 2\alpha_k(1-\alpha_k\lambda)\langle w^k, g^k\rangle + \alpha_k^2\|g_k\|_2^2, \\ =& (1-\alpha_k\lambda)^2\gamma^k - 2\alpha_k(1-\alpha_k\lambda)\beta^k\langle v^k, g^k\rangle + \alpha_k\|g^k\|_2^2, \\ =& (1-\alpha_k\lambda)^2\gamma^k - 2\alpha_k\beta^{k+1/2}\langle v^k, g^k\rangle + \alpha_k\|g^k\|_2^2.\end{aligned}$$

This now gives $\beta^{k+1/2} \sim O(1)$, $\gamma^{k+1/2} \sim O(k)$ and $\nu^{k+1/2} \sim O(1)$. The next step,

$$\beta^{k+1} = \beta^{k+1/2}, \quad \gamma^{k+1} = \gamma^{k+1/2},$$
$$\nu^{k+1} = \begin{cases} \nu^{k+1/2} - \frac{\alpha_k}{\beta^{k+1/2}} g^k & \text{if } \sqrt{\gamma^{k+1/2}} \leq \tau \\ \frac{\tau}{\sqrt{\gamma^{k+1/2}}}\left(\nu^{k+1/2} - \frac{\alpha_k}{\beta^{k+1/2}} g^k\right) & \text{if } \sqrt{\gamma^{k+1/2}} > \tau \end{cases}$$

Where $\beta^{k+1} \sim O(1)$, $\gamma^{k+1} \sim O(1)$ and $\nu^{k+1} \sim O(k)$. In comparison to $w^{k+1/2} \sim O(d)$ and $w^{k+1} \sim O(k^2)$, this is much faster and allows for sparsity to be used.

[2] We can obtain a value for τ for any L2-regularized problem where the loss is bounded below, by using that $f(w^*) + \frac{\lambda}{2}\|w^*\|^2 \leq f(0)$ and replacing $f(w^*)$ with its lower bound (which can be 0 for SVMs).

2 Computation Questions

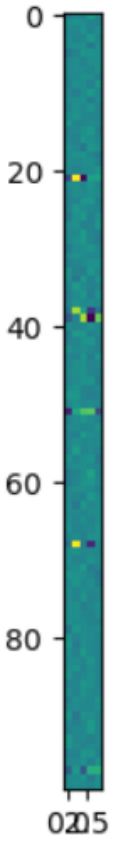
2.1 Proximal-Gradient

If you run the demo *example_group.jl*, 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, *logRegSoftmaxL2*, 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 validation error achieved with $\lambda = 10$ (which is the 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.

   Answer:
   TrainError: 0.006
   ValidError: 0.274
   Number of original parameters: 500
   Number of nonzero parameters: 100

Answer: Sparsity pattern and Code of logRegSoftmaxL2.jl




```
# Multi-class softmax version (assumes y_i in {1,2,...,k})
function logRegSoftmaxL2(X,y,lambda)
    (n,d) = size(X)
    k = maximum(y)
    # Each column of 'w' will be a logistic regression classifier
    W = zeros(d,k)
    funObj(w) = softmaxL2Obj(w,X,y,k,lambda)
    W[:] = findMin(funObj,W[:],derivativeCheck=true,maxIter=500)
    # Make linear prediction function
    predict(Xhat) = mapslices(indmax,Xhat*W,2)
    return LinearModel(predict,W)
end
function softmaxL2Obj(w,X,y,k,lambda)
    #computes objective function and Gradient
    (n,d) = size(X)
    W = reshape(w,d,k)
    XW = X*W
    Z = sum(exp.(XW),2)
    nll = 0
    G = zeros(d,k)
    for i in 1:n
        nll += -XW[i,y[i]] + log(Z[i])
        pVals = exp.(XW[i,:])./Z[i]
        for c in 1:k
            G[:,c] += X[i,:]*(pVals[c] - (y[i] == c))
        end
    end
    #add the terms for the L2 regularization
    G = G + lambda*W
    nll += lambda/2*sum(W.^2)
    return (nll,reshape(G,d*k,1))
end
```

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, *logRegSoftmaxL1*, that fits a multi-class logistic regression model with L1-regularization. You can use the function *findMinL1*, 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.

   Answer: We obtained the best results for $\lambda = 8.0$.
   TrainError: 0.03
   ValidError: 0.072
   Number of original parameters: 45
   Number of nonzero parameters: 28

   For the value of lambda from the provious problem: $\lambda = 10$.
   TrainError: 0.048
   ValidError: 0.08
   Number of original parameters: 35
   Number of nonzero parameters: 19

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 (*logRegSoftmaxGL1* 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.

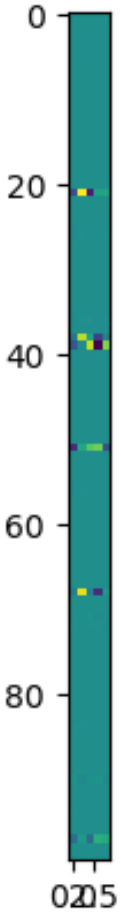
   Answer: There are many different ways to set up the group assignment in this problem. For $\lambda = 10$ and by assigning each row of W to a separate group ($d = 100$ groups) we obtained
   TrainError: 0.022
   ValidError: 0.054
   Number of original parameters: 115
   Number of nonzero parameters: 23

   To test our Algorithm we assigned each element in W a unique group and set $\lambda = 8$. This gave exactly the same results as in 2.1.2, which made us very very happy. The following piece of code shows how to set up a group that one can feed our algorithm:

   Answer: Setup of the group matrix

```
#setup the group for the groupL1regularization
gro = ones(d,k)
for j=1:k
    #set up the groups rowwise
    #ALL different groups
    #gro[:,j]=linspace(1+d*(j-1),d*j,d)
    #ROWWISE same groups
    gro[:,j]=linspace(1,d,d)
    #COLUMNWISE same groups
    #gro[:,j]=j*ones(1,100)
end
model = softmaxClassiferGL1(X,y,lambda,gro)
```

Answer: Sparsity pattern of W and code of proxGradGroupL1.jl




```julia
function proxGradGroupL1(funObj,w,gro,lambda;maxIter=100,epsilon=1e-2)
    # Evalluate the intial objective and gradient
    (f,g) = funObj(w)
    # Initial step size and sufficient decrease parameter
    (gamma,alpha) = [1e-4,1]
    for i in 1:maxIter
        # Gradient step on smoooth part
        wNew = w - alpha*g
        # Proximal step
        wNew = proxStep(wNew ,alpha*lambda,gro)
        (fNew,gNew) = funObj(wNew)
        # Decrease the step-size if we increased the function
        gtd = dot(g,wNew-w)
        while fNew + lambda*groupL1Norm(wNew,gro) > f + lambda*groupL1Norm(w,gro) - gamma*alpha*gtd
            @printf("Backtracking\n")
            alpha /= 2
            # Try out the smaller step-size
            wNew = w - alpha*g
            wNew = proxStep(wNew ,alpha*lambda,gro)
            (fNew,gNew) = funObj(wNew)
        end
        # Guess the step-size for the next iteration
        y = gNew - g
        alpha *= -dot(y,g)/dot(y,y)
        # Sanity check on the step-size
        if (!isfinitereal(alpha)) | (alpha < 1e-10) | (alpha > 1e10)
            alpha = 1
        end
        # Accept the new parameters/function/gradient
        (w,f,g)=[wNew,fNew,gNew]
        # Print out some diagnostics
        optCond = groupL1Norm(w-proxStep(w-g ,alpha*lambda,gro),gro)
        @printf("%6d %15.5e %15.5e %15.5e\n",i,alpha,f+lambda*groupL1Norm(w,gro),optCond)
        # We want to stop if the gradient is really small
        if optCond < epsilon
            @printf("Problem solved up to optimality tolerance\n")
            return w
        end
    end
    @printf("Reached maximum number of iterations\n")
    return w
end
```

Answer: Code of proxStep and groupL1Norm

```julia
function  proxStep(w,threshold,gro)
    maxgro = maximum(gro)
    for j in 1:maxgro
        ind = find(x->x==j,gro)
        groupnorm = norm(w[ind])
        if groupnorm !=0
            w[ind] = w[ind]/groupnorm * max(0,groupnorm-threshold);
        end
    end
    return w
end

function groupL1Norm(w,gro)
    maxgro = maximum(gro)
    GL1 = 0
    for j in 1:maxgro
        ind = find(x->x==j,gro)
        GL1 += norm(w[ind])
    end
    return GL1
end
```

2.2 Coordinate Optimization

The function *example_CD.jl* 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 $\frac{1}{4}\max_j\{\|x_j\|^2\} + \lambda$, where x_j is column j of the matrix X). Hand in your code and report the final function value and total time.

   Answer:

```
## L = .25*maximum(eigenvalues) + lambda;
L = 0.25* maximum( sum(X.^2,1) ) + lambda

# Start running coordinate descent
w_old = copy(w);
xw = zeros(n,1)
for k in 1:maxPasses*d

    # Choose variable to update 'j'
    j = rand(1:d)

    # Compute partial derivative 'g_j'
##    yXw = y.*(X*w);
    yXw = y.*xw
    sigmoid = 1./(1+exp.(-yXw));
## g = -X'*(y.*(1-sigmoid)) + lambda*w;
## g_j = g[j];


    g_j = -X[:,j]'*(y.*(1-sigmoid)) + lambda*w[j];
    g_j = g_j[1]
        # Update variable
    w_oj = copy(w[j])
    w[j] -= (1/L)*g_j;
    xw = xw + X[:,j] * (w[j]-w_oj)

    # Check for lack of progress after each "pass"
    if mod(k,d) == 0
```

   Passes = 500,
   function = 1.4728e+02,
   change = 0.0004
   elapsed time: 9.647018359 seconds

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* (in *misc.jl*) to sample from a discrete distribution given the probability mass function. Hand in your code, and report the final function value as well as the number of passes.

   Answer:

```
# Start timer
tic()

# Compute Lipschitz constant of 'f'
L = 0.25*( sum(X.^2,1) ) + lambda

# Start running coordinate descent
w_old = copy(w);
xw = zeros(n,1)
for k in 1:maxPasses*d

    # Choose variable to update 'j'
    j = sampleDiscrete(L/sum(L))

    # Compute partial derivative 'g_j'
##    yXw = y.*(X*w);
    yXw = y.*xw
    sigmoid = 1./(1+exp.(-yXw));
## g = -X'*(y.*(1-sigmoid)) + lambda*w;
## g_j = g[j];


    g_j = -X[:,j]'*(y.*(1-sigmoid)) + lambda*w[j];
    g_j = g_j[1]
        # Update variable
    w_oj = copy(w[j])
    w[j] -= (1/L[j])*g_j;
    xw = xw + X[:,j] * (w[j]-w_oj)
```

Passes = 277,
function = 1.4428e+02,
change = 0.0001
elapsed time: 6.717980036 seconds

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 $1/L_{j_t}$.

   Answer: Passes = 143,
   function = 1.4091e+02,
   change = 0.0001
   elapsed time: 2.829235952 seconds

4. Suppose that when we use a step-size of $1/L_{j_t}$, we see that uniform sampling outperforms Lipschitz sampling. Why could this be consistent with the bounds we've shown?

   Answer: If we perform a weighted sampling such than the coordinate with the highest Lipschitz constants have the highest probability of occurring, then the smallest step sizes among $(1/L_j)_{j=1,\dots,d}$ have the highest probability. That means we are approaching the minimum slowly. In the contrary, taking a uniform sampling will have a larger expected step length which means that we could approach the minimum faster.

2.3 Stochastic Gradient

If you run the demo *example_SG.jl*, 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 in other high-level languages (like R/Matlab/Python) this demo would run really slowly so you would need to write the inner loop in a low-level language like C, but in Julia you can directly write the stochastic gradient code and have it run fast.

Unfortunately, despite Julia making this code run fast compared to other high-level languages, the performance of this stochastic gradient method is atrocious. It often goes to areas of the parameter space where 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 if your step-size is small enough.)

1. Although $\alpha_t = 1/\lambda t$ 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 sequence to see if you can get better performance. Report the step-size sequence that you found gave the best performance, and the objective function value obtained by this strategy for one run.

   Answer: We didn't have much success having a decaying sequence of step sizes since we only pass through the data set once. Instead we chose a constant step size of $\alpha_k = 50/n = .001$ which gave $f = 2.7204e+04$ for a single pass.

   Answer: Code of example_SG_modified1.jl

```julia
# Load X and y variable
using JLD
data = load("quantum.jld")
(X,y) = (data["X"],data["y"])
# Add bias variable, initialize w, set regularization and optimization paramet
(n,d) = size(X)
lambda = 1
# Initialize
maxPasses = 1
progTol = 1e-4
verbose = true
w = zeros(d,1)
lambda_i = lambda/n # Regularization for individual example in expectation

# Start running stochastic gradient
w_old = copy(w);
for k in 1:maxPasses*n
    # Choose example to update 'i'
    i = rand(1:n)

    # Compute gradient for example 'i'
    r_i = -y[i]/(1+exp(y[i]*dot(w,X[i,:])))
    g_i = r_i*X[i,:] + (lambda_i)*w

    # Choose the step-size
    alpha = 50/n

    # Take thes stochastic gradient step
    w -= alpha*g_i

    # Check for lack of progress after each "pass"
    if mod(k,n) == 0
        yXw = y.*(X*w)
        f = sum(log.(1 + exp.(-y.*(X*w)))) + (lambda/2)norm(w)^2
        delta = norm(w-w_old,Inf);
        if verbose
            @printf("Passes = %d, function = %.4e, change = %.4f\n",k/n,f,delt
        end
        if delta < progTol
            @printf("Parameters changed by less than progTol on pass\n");
            break;
        end
        w_old = copy(w);
    end
end
```

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.)

Answer: Choosing $\alpha_k = 0.001663$ from optimizing over the parameter, we implement an average over only the second half of the iterations of w^k. This gives $f = 2.7155e + 04$ for a single pass.

Answer: Code of example_SG_modified2.jl

```
# Load X and y variable
using JLD
data = load("quantum.jld")
(X,y) = (data["X"],data["y"])
# Add bias variable, initialize w, set regularization and optimization parame
(n,d) = size(X)
lambda = 1
# Initialize
maxPasses = 1
progTol = 1e-4
verbose = true
w = zeros(d,1)
wbar = w
lambda_i = lambda/n # Regularization for individual example in expectation

# Start running stochastic gradient
w_old = copy(w);
m=maxPasses*n

for k in 1:m

    # Choose example to update 'i'
    i = rand(1:n)

    # Compute gradient for example 'i'
    r_i = -y[i]/(1+exp(y[i]*dot(w,X[i,:])))
    g_i = r_i*X[i,:] + (lambda_i)*w

    # Choose the step-size
    alpha = 0.001663

    # Take thes stochastic gradient step
    w -= alpha*g_i
    # Proportionally weighted average
    if k>=n/2
        wbar += 2*w/n
    end

    # Check for lack of progress after each "pass"
    if mod(k,n) == 0
        yXw = y.*(X*w)
        f = sum(log.(1 + exp.(-y.*(X*w)))) + (lambda/2)norm(w)^2
        fbar = sum(log.(1 + exp.(-y.*(X*wbar)))) + (lambda/2)norm(wbar)^2
        delta = norm(w-w_old,Inf);
        if verbose
            @printf("Step size = %f, function = %.4e, change = %.4f, average
        end
        if delta < progTol
            @printf("Parameters changed by less than progTol on pass\n");
            break;
        end
        w_old = copy(w);
    end
end
```

3. A popular variation on stochastic is AdaGrad, which uses the iteration

$$w^{k+1} = w^k - \alpha_k D_k \nabla f(w^k),$$

where the element in position (j, j) of the diagonal matrix D_k is given by $1/\sqrt{\delta + \sum_{k'=0}^{k}(\nabla_j f_{i_{k'}}(w^{k'}))^2}$. Here, i_k is the example i selected on iteration k and ∇_j denotes element j of the gradient (and in AdaGrad we typically 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.
The best parameters found through a double line search were: $\alpha_k = 0.031250$ and $\delta = 0.031250$ which gives the function value on the first pass $f = 2.7206e + 04$.

Answer: Code of example_SG_modified3.jl

```
# Load X and y variable
using JLD
data = load("quantum.jld")
(X,y) = (data["X"],data["y"])

# Add bias variable, initialize w, set regularization and optimization parame
(n,d) = size(X)
lambda = 1
D = zeros(d,d)
sumvec = zeros(d,1)
delta=0.031250
# Initialize
maxPasses = 1
progTol = 1e-4
verbose = true
w = zeros(d,1)
lambda_i = lambda/n # Regularization for individual example in expectation

# Start running stochastic gradient
w_old = copy(w);
for k in 1:maxPasses*n

    # Choose example to update 'i'
    i = rand(1:n)

    # Compute gradient for example 'i'
    r_i = -y[i]/(1+exp(y[i]*dot(w,X[i,:])))
    g_i = r_i*X[i,:] + (lambda_i)*w

    # Store these gradients
    sumvec += g_i.^2
    for j in 1:d
        D[j,j] = 1/sqrt(delta+sumvec[j])
    end

    # Choose the step-size
    alpha=0.031250
    # Take thes stochastic gradient step
    w -= alpha*D*g_i
    # Check for lack of progress after each "pass"
    if mod(k,n) == 0
        yXw = y.*(X*w)
        f = sum(log.(1 + exp.(-y.*(X*w)))) + (lambda/2)norm(w)^2
        delta = norm(w-w_old,Inf);
        if verbose
            @printf("Passes = %d, function = %.4e, change = %.4f\n",k/n,f,del
        end
        if delta < progTol
            @printf("Parameters changed by less than progTol on pass\n");
            break;
        end
        w_old = copy(w);
    end
end
```

4. Implement the SAG algorithm with a step-size of $1/L$, where L is the maximum Lipschitz constant across the training examples ($L = \frac{1}{4}\max_i\{\|x^i\|^2\} + \lambda$). Hand in your code and again report the performance for one run.

   Answer: For one run: $f = 4.0384e+04$. This is terrible performance but consider multiple runs. For 10 runs: $f = 2.7073e+04$. For multiple passes, this is performing fantastically!

   Answer: Code of example_SG_modified4.jl

```
# Load X and y variable
using JLD
data = load("quantum.jld")
(X,y) = (data["X"],data["y"])
# Add bias variable, initialize w, set regularization and optimization paramet
(n,d) = size(X)
lambda = 1
# Initialize
maxPasses = 1
progTol = 1e-4
verbose = true
w = zeros(d,1)
v = zeros(n,d)
g = zeros(d,1)
lambda_i = lambda/n # Regularization for individual example in expectation
# Start running stochastic gradient
w_old = copy(w);
norms=zeros(n,1)
# Find lipschitz
for i in 1:n
    norms = norm(X[i,:])^2
end
L=.25*max(norms)+lambda
for k in 1:n*maxPasses
    # Choose example to update 'i'
    i = rand(1:n)
    # Compute gradient for example 'i'
    r_i = -y[i]/(1+exp(y[i]*dot(w,X[i,:])))
    g_i = r_i*X[i,:] + (lambda_i)*w
    # Store these gradients
    g = g-v[i,:]+g_i
    v[i,:] = g_i
    # Choose the step-size
    alpha = 1/L

    # Take thes stochastic gradient step
    w -= alpha*g/(maxPasses*n)

   # Check for lack of progress after each "pass"
    if mod(k,n) == 0
        yXw = y.*(X*w)
        f = sum(log.(1 + exp.(-yXw))) + (lambda/2)*norm(w)^2
        change = norm(w-w_old,Inf);
        if verbose
            @printf("pass =%d, alpha = %f, function = %.4e, change = %.4f\n",k
        end
        if change < progTol
            @printf("Parameters changed by less than progTol on pass\n");
            break;
        end
        w_old = copy(w);
    end
end
```

3 Very-Short Answer Questions

Give a short and concise 1-sentence answer to the below questions.

1. What is a situation where we can violate the golden rule on our test set, and still have it be a reasonable approximation of test error?

   Answer: In transductive learning, we can violate the golden rule to create more labels and therefore perform supervised learning methods rather than unsupervised. This violates our rule as we are biasing our validation set but this is to get faster techniques or for accuracy.

2. Do convex functions necessarily have a minima?

   Answer: No. Consider $f(x) = x$ and note that $\frac{d^2}{dx^2} f(x) \geqslant 0$.

3. What is the purpose of the descent lemma?

   Answer: We use it to obtain convergence estimates for gradient descent for differentiable functions with Lipschitz continuous gradients. In particular, we used it to obtain the progress bound.

4. Why did we show that the gradient norm converges to 0 when we run gradient descent? (As opposed to showing that gradient descent finds the global optimum.)

   Answer: Showing that the norm of the gradient converges to zero justifies that the algorithm converges. This might not be the global minimum as it could be just a local minimum. Another possible case is that our function is constant. The statement that we converge to a global minimum wouldn't make sense. However, for a strictly convex function convergence of the gradient norm to zero is equivalent to convergence to a local minimum.

5. What is the relationship between the projected-gradient method and the proximal-gradient method?

   Answer: Projected gradient is a special case of proximal gradient, the proximal operator is solving a smooth+non-smooth problem and if we just allow the smooth part to forbid the solution from happening outside a region, we get the projected gradient method.

6. Why do we prefer the proximal-gradient method over the subgradient method?

   Answer: The proximal gradient method will give a more sparse solution than the subgradient method. Furthermore, the subgradient method is often slower than proximal gradient.

7. Why did we say in 340 that regularizing by the L2-norm doesn't give sparsity, and now we're saying we can use the L2-norm to give sparse solutions?

   Answer: The L2-norm squared won't give sparsity because its smooth around 0 and thus would give a sparse solution with probability 0. The L2-norm has a cusp at the origin and therefore can give a sparse solution with positive probability.

8. What is an example of a pattern that can be enforced with structured sparsity?

   Answer: We can create a k-diagonal matrix that has bandwidth k. This can speed up calculations massively if we have a structured weight matrix (see Discrete Wavelet Transforms). The example from the lecture is group L1 regularization. Here we force certain entries in our matrix W to be zero or non zero *simultaneously*.

9. Under what condition do we prefer to use coordinate optimization over gradient descent?

   Answer: We would like our function to be separable. Computing the partial derivatives has to be d-times cheaper. Also, if we have vastly different Lipschitz constants in each coordinate then coordinate optimization will perform much better.

10. For a machine learning objective that is the sum over n training examples, why should we never use the subgradient method?

    Answer: Rather than computing the subgradient for each example, we could use stochastic subgradient instead to speed up performance while still maintaining the convergence rate. The difference is cost and subgradient will cost a lot for many examples.

11. What is the advantage of SAG over stochastic gradient methods without a memory?

    Answer: Although SAG requires memory, this means we only need to compute one gradient per iteration and hence it is very cheap to run SAG.