CPSC 540 Assignment 2 (due February 1 at midnight)

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): Erik Arne Huso, Vetle Birkeland Huglen, Elias Loona Myklebust
2. Student ID(s): 91227694, 13335773, 19750611

1 Calculation Questions

1.1 Convexity

1. We compute the gradient of the function as $\nabla f = X^T vXw - X^T Vy + \lambda w$ and thus we get Hessian $\nabla^2 f = X^T VX + \lambda I$. The hessian is positive semidefinite since for any vector v we have

$$\begin{aligned} v^T(X^T VX\lambda I)v &= v^T VXVv + \lambda v^T v \\ &= \|V^{\frac{1}{2}}Xv\| + \lambda\|v^T v\| \\ &\geq 0 \end{aligned}$$

   using that V is diagonal with positive entries. So the function is convex.

2. Taking the gradient we get $\nabla f(w) = -y^T X + d$, where d is a vector with elements $d_j = \sum_i \exp(w^T x^i)x_j^i$. Taking the derivative again we now get $\nabla^2 f(w) = D$, where D is a matrix with elements $D_{jk} = \sum_i \exp(w^T x^i)x_j^i x_k^i$. To show that this matrix (the Hessian) is positive semidefinite we take a vector v and look at

$$\begin{aligned} v^T Dv &= \sum_j \sum_k D_{jk} v_j v_k \\ &= \sum_j \sum_k \left(\sum_i \exp(w^T x^i)x_j^i x_k^i \right) v_j v_k \\ &= \sum_i \exp(w^T x^i) \sum_j x_j^i v_j \sum_k x_k^i v_k \\ &= \sum_i \exp(w^T x^i) \left(\sum_j x_j^i v_j \right)^2 \\ &\geq 0 \end{aligned}$$

   since the exponential function is always greater than 0 and taking the square of the last factor is greater than or equal to 0. Thus the function is convex.

3. We have

$$\begin{aligned} f(\lambda x + (1-\lambda y)) &= \max_j |L_j(\lambda x_j + (1-\lambda)y_j| \\ &= \max_j |L_j \lambda x_j + L_j(1-\lambda)y_j| \\ &\leq \max_j |L_j \lambda x_j| + |L_j(1-\lambda)y_j| \quad \text{(by the triangle inequality)} \\ &\leq \lambda \max_j |L_j x_j| + (1-\lambda) \max_j |L_j y_j| \\ &= \lambda f(x) + (1-\lambda) f(y) \end{aligned}$$

   so the function is convex.

4. In general p-norms are convex, and both multiplication by a non-negative scalar and addition of convex functions preserve convexity, so the function must be convex.

5. We can rewrite the function as $f(w) = \sum_{i=1}^{N} \max\{0, \max\{-(w^\top x_i - y_i) - \epsilon, (w^\top x_i - y_i) - \epsilon\}\} + \frac{\lambda}{2}\|w\|_2^2$. We know that taking the max of two convex functions preserves convexity. For the inner max we are taking the maximum of two linear functions which must be convex. Thus the outer max is also convex. Since sums preserve convexity and the last term is convex by convexity of norms and multiplication by non-negative numbers, the whole function must be convex.

6. The set of vectors x that satisfy the linear constraints $Ax \leq b$ is convex (for x,y in this set take a convex combination and obtain $A(\lambda x + (1-\lambda)y) = \lambda Ax + (1-\lambda)Ay \leq \lambda b + (1-\lambda)b = b$). By one of the definitions of convex functions we need $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$. If x and y are both in the previously described set, then all of these function values will be zero and the inequality holds. For the function value on the left hand side to be ∞, at least one of x and y need to lie outside the set and thus at least one of the function values on the right hand side must also be ∞ and thus the inequality holds in this case also.

1.2 Convergence of Gradient Descent

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

1. If we require that $\|\nabla f(w^k)\|^2 \leq 2L(f(w^0) - f^*)/t^{4/3} \leq \epsilon$, then by rearranging we get that

$$\begin{aligned} & t \geq 2L(f(w^0) - f^*)^{\frac{3}{4}}/\epsilon^{\frac{3}{4}} \\ \implies & t = O(1/\epsilon^{3/4}) \end{aligned}$$

2. We use the fact that we now must have $L \leq L^k \leq 2L$, so the gradient is also L^k-Lipschitz, and by strong convexity we have that the inequality $-\|\nabla f(x^k)\|^2/2 \leq -\mu[f(w^k) - f(w^*)]$ must be satisfied. Starting with the given inequality and using these properties we get

$$\begin{aligned} f(w^k) &\leq f(w^{k-1}) - \frac{1}{2L^k}\|\nabla f(w^{k-1)}\|^2 \\ &\leq f(w^{k-1}) - \frac{1}{2 \cdot 2L}\|\nabla f(w^{k-1)}\|^2 \\ &\leq f(w^{k-1}) - \frac{\mu}{2L}\left[f(w^{k-1}) - f(w^*)\right] \end{aligned}$$

Subtracting $f(w^*)$ on both sides yields

$$\begin{aligned} f(w^k) - f(w^*) &\leq f(w^{k-1}) - f(w^*) - \frac{\mu}{2L}\left[f(w^{k-1}) - f(w^*)\right] \\ &= \left(1 - \frac{\mu}{2L}\right)\left[f(w^{k-1}) - f(w^*)\right] \\ \implies f(w^k) - f(w^*) &\leq \left(1 - \frac{\mu}{2L}\right)^k \left[f(w^0) - f(w^*)\right] \end{aligned}$$

3.

4. By the μ-strong convexity we have that for v, w we have $f(v) \geq f(w) + \nabla f(w)^T(v-w) + \frac{\mu}{2}\|v-w\|^2$. Setting $v = w^k$ and $w = w^*$ and using that $\nabla f(w^*) = 0$ we get, after rearranging, that

$$\begin{aligned} \|w^k - w^*\| &\leq \left(\frac{2}{\mu}\left[f(w^k) - f(w^*)\right]\right)^{1/2} \\ \implies \|w^k - w^*\| &= O(\rho^{k/2}) \end{aligned}$$

1.3 Beyond Gradient Descent

1. We have

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

Then, inserting the first equation into the last, we obtain:

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

Expanding the square in the expression and dividing everything by α_k yields

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

By noticing that $\alpha_k (w^k)^2$ is a constant that does not influence the minimization problem, we can replace this with another constant, $f(w)$ to finally obtain

$$w^{k+1} \in \operatorname*{argmin}_{v \in \mathbb{R}^d} \left\{ f(w^k) + \nabla f(w^k)^\top (v - w^k) + \frac{1}{2\alpha_k}\|v - w^k\|^2 + r(v) \right\}. \tag{3}$$

2. We have

$$f(W) = \sum_{i=1}^{n} \sum_{c \neq y^i} [1 - w_{y^i}^\top x^i + w_c^\top x^i]^+ + \frac{\lambda}{2}\|W\|_F^2,$$

where $[\gamma]^+$ sets negative values to zero. If we focus on the inner sum of the loss function, for the cases where $c \neq y^i$, we get the gradient to be

$$-x^i \mathbb{1}(1 - w_{y^i}^T x^i + w_c^T x^i > 0).$$

This gradient of the regularizer is

$$\lambda \|W\|_F,$$

which yields that the gradient of $f(w)$ is:

$$\nabla f(w) = \sum_{i=1}^{n} \sum_{c \neq y^i} [-x^i \mathbb{1}(1 - w_{y^i}^T x^i + w_c^T x^i > 0)] + \lambda \|W\|_F. \tag{4}$$

3. Using the given information, we have f bounded below and where ∇f is Lipschitz continuous in the ∞-norm, so that

$$f(v) \leq f(u) + \nabla f(u)^\top (v - u) + \frac{L_\infty}{2} \|v - u\|_\infty^2,$$

for all v and w and some L_∞. In addition, we have

$$w^{k+1} = w^k - \frac{\|\nabla f(w^k)\|_1}{L_\infty} \text{sign}(\nabla f(w^k)).$$

For ease of notation, we define $f(w^k) := f_k$ and so on. Letting $v = w^{k+1}$ and $u = w^k$, and inserting into the first equation, we get

$$\begin{aligned} f(w^{k+1}) &\leq f_k + \nabla f_k^T (w^{k+1} - w^k) + \frac{L_\infty}{2} \|w^{k+1} - w^k\|_\infty^2 \\ &= f_k - \nabla f_k^T \frac{\|\nabla f_k\|_1}{L_\infty} \text{sign}(\nabla f_k) + \frac{L_\infty}{2} \| - \frac{\|\nabla f_k\|_1}{L_\infty} \text{sign}(\nabla f_k)\|_\infty^2 \\ &= f_k - \frac{1}{L_\infty} \|\nabla f_k\|_1 \nabla f_k^T \text{sign}(\nabla f_k) + \frac{1}{2L_\infty} \|\nabla f_k\|_1^2 \\ &= f_k - \frac{1}{2L_\infty} \|\nabla f_k\|_1^2. \end{aligned}$$

Rearranging the terms and shifting iterations one step back, we have

$$\|\nabla f_{k-1}\|_1^2 \leq 2L_\infty (f_{k-1} - f_k). \tag{5}$$

If we sum over the t iterations on both sides, the right hand side becomes $(f(w^0) - f(w^t)$, because of the telescoping effect of the series. As left hand side does not depend on k, we get

$$t \min_{k \in (0,\ldots,t-1)} \|\nabla f(w^k)\|_1^2 \leq 2L_\infty (f(w^0) - f(w^*), \tag{6}$$

since we further utilize that $f(w^t) \geq f^*$ when f^* is optimal solution. Then, if $\|\nabla f(w^k)\|^2 \leq \epsilon$, we get

$$t \geq \frac{2L_\infty (f(w^0) - f(w^*)}{\epsilon}, \tag{7}$$

which yields a cost of $O(1/\epsilon)$.

2 Computation Questions

2.1 Proximal Gradient

2.1.1 Softmax with L2-regularization

A model using $\lambda = 10$ and Softmax with L2-regularization achieved validation error of 0.3, slightly larger than without the regularization. This model used all 100 original features, and none of the 500 model parameters were zero.

```
115
116
117 function softmaxObjL2(w,X,y,k,lambda)
118     (n,d) = size(X)
119
120     W = reshape(w,d,k)
121
122     XW = X*W
123     Z = sum(exp.(XW),dims=2)
124
125     nll = 0
126     G = zeros(d,k)
127     for i in 1:n
128         nll += -XW[i,y[i]] + log(Z[i])
129
130         pVals = exp.(XW[i,:])./Z[i]
131         for c in 1:k
132             G[:,c] += X[i,:]*(pVals[c] - (y[i] == c))
133         end
134     end
135
136     # Add L2 regularization to objective function
137     nll += lambda*sum(W.*W)
138
139     # Add L2 regularization to objective gradient
140     G += 2*lambda*W
141
142     return (nll,reshape(G,d*k,1))
143 end
144
145
```

Figure 1: 2.1.1: Implementation of Softmax with L2-regularization.

2.1.2 Softmax with L1-regularization

When instead using L1-regularization, this model achieved a validation error of 0.08, significantly better than both non-regularized and L2-regularized Softmax. 19 of the original 100 features were used, and the model consisted of 35 non-zero parameters.

2.1.3 Softmax with Group L1-regularization

The best performance was achieved using Softmax with group L1-regularization. With $\lambda = 10$ the validation error was 0.044, almost halving the error from the previous model. The number of non-zero parameters were reduced to 30, and these consisted of only 6 original features, meaning the model discarded 94% of the available explanatory variables.

```
function softmaxClassiferGL1(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) = softmaxObj(w,X,y,k)

    W[:] = proxGradGroupL1(funObj,W[:],d,lambda,maxIter=500)

    # Make linear prediction function
    predict(Xhat) = mapslices(argmax,Xhat*W,dims=2)

    return LinearModel(predict,W)

end
```

Figure 2: 2.1.3: Model function.

```
function proxGradGroupL1(funObj,w,d,lambda;maxIter=100)

    alpha = 1
    gamma = 1e-4
    (f,g) = funObj(w)

    for i in 1:maxIter
        # Gradient step on smoooth part
        wNew = w - alpha*g

        # Groupwise proximal step on non-smooth part
        for i in 1:d
            wNew[i:d:end] = wNew[i:d:end]*max(0,norm(wNew[i:d:end],2)-alpha*lambda)/norm(wNew[i:d:end],2)
        end

        # Setting new objective values
        (fNew,gNew) = funObj(wNew)

        # Reduce step size if function value increases
        while fNew > f
            alpha /= 2
            # Gradient step on smoooth part
            wNew = w - alpha*g

            # Groupwise proximal step on non-smooth part
            for i in 1:d
                wNew[i:d:end] = wNew[i:d:end]*max(0,norm(wNew[i:d:end],2)-alpha*lambda)/norm(wNew[i:d:end],2)
            end

            # Setting new objective values
            (fNew,gNew) = funObj(wNew)
        end

        # Accept new parameters
        w = wNew
        f = fNew
        g = gNew
    end

    return w
end
```

Figure 3: 2.1.3: Implementation of Proximate Gradient method. Loosely based on the already implemented `findMinL1`-function.

2.2 Coordinate Optimiziation

1. After modifying the code in order to get a runtime of $O(nd\frac{L_c}{\mu}log(1/\epsilon)$, the total elapse of the method once takes about 2.69 seconds, 25249 iterations and 250 passes. This was obtained by changing the

Lipschitz constant from L_f to L_c, and modifying the iterations so that the gradient is calculated by the chosen index initially, instead of picking the index gradient after calculating the complete gradient.

2. Instead of using a constant step size L_c, we can choose a variying step size using backtracking. By implementing backtracking line search utilizing the Armijo condition for one gradient i at iteration k

$$f(x^k - \alpha \nabla_{i_k} f(x^k) e_{i_k}) < f(x_k), \tag{8}$$

choosing α to satisfy the above inequality. e_j is a vector with value 1 at index j and the rest zeros. With this method, the number of iterations is reduced to about 7473 and the number of passes to 74, while the elapsed time is kept at approximately the same level, 2.69 seconds.

Code for Questions 2.2.1 and 2.2.2:

```
# Load X and y variable
using JLD, Printf, LinearAlgebra



data = load("binaryData.jld")
(X,y) = (data["X"],data["y"])

# Add bias variable, initialize w, set regularization and optimization parameters
(n,d) = size(X)
X = [ones(n,1) X]
d += 1
w = zeros(d,1)
lambda = 1
maxPasses = 500
progTol = 1e-4
verbose = true

## Run and time coordinate descent to minimize L2-regularization logistic loss

# Start timer
time_start = time_ns()

# Compute Lipschitz constant of 'f'
#sd = eigen(X'X)
#L = maximum(sd.values) + lambda;

#compute new lipschitz constant. Q2.2.1

Lc = maximum(sum(X.^2, dims=1)) + lambda

# Start running coordinate descent
w_old = copy(w);
iterations = 0

r = X*w - y

for k in 1:maxPasses*d

    # Choose variable to update 'j'
```

```
j = rand(1:d)

# Compute partial derivative 'g_j'
"""
r = X*w - y
g = X'*r + lambda * w
g_j = g[j];
"""
#Reduce runtime of iteration
global r
g_j = dot(X[:,j],r)

#Backtracking line search
alpha = 1/100
c = 10^0 #Arbitrary constant

fxk = (1/2)*norm(r)^2 + lambda/2 * norm(w)^2
e = zeros(d)
e[j] = 1
while true
    r_new = r - X[:,j]*alpha * c * g_j
    #Armijo condition
    if ((1/2)*norm(r_new)^2 + lambda/2 * norm(w - alpha * c * g_j * e)^2 > fxk)
        alpha /= 4
    else
        break;
    end
end

# Update variable
#w[j] -= (1/Lc)*(g_j + lambda*w[j]);

#Update variable with varying step size alpha
w[j] -= alpha*(g_j + lambda*w[j]);

#Update r

r = r - X[:,j]*(alpha)*g_j #Updating with varying alpha
#r = r - X[:,j]*(1/Lc)*g_j #Updating with Lc

# Check for lack of progress after each "pass"
# - Turn off computing 'f' and printing progress if timing is crucial
if mod(k,d) == 0
    r = X*w - y
    f = (1/2)norm(r)^2 + (lambda/2)norm(w)^2
    delta = norm(w-w_old,Inf);
    if verbose
        @printf("Passes = %d, function = %.4e, change = %.4f\n",k/d,f,delta);
    end
    if delta < progTol
        @printf("Parameters changed by less than progTol on pass\n");
```

```
            break;
        end
        global w_old = copy(w);
    end
    global iterations += 1
end

# End timer
@printf("Seconds = %f\n",(time_ns()-time_start)/1.0e9)
@printf("Iterations = %f\n", iterations)
```

2.3 Stochastic Gradient

1. The step-size sequence that gave the best results seemed to be $\alpha_k = 1/k^{2/3}$. With this sequence we got a function value of $f = 2.7088 \cdot 10^4$ after 10 passes.
2. With averaging over the 100 last iterations and using the same step-size sequence (this was found to be best), we got a (very) slightly slower runtime and function value of $f = 2.7098 \cdot 10^4$ after 10 passes.
3. With AdaGrad the best step-size sequence we found was $alpha_k = 1/\sqrt{k}$. The performance was slightly worse. We got a function value of $f = 2.7838 \cdot 10^4$ after 10 passes. The code use is the following

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

# Add bias variable, initialize w, set regularization and optimization parameters
(n,d) = size(X)
lambda = 1
delta = 1

# Initialize
maxPasses = 10
progTol = 1e-4
verbose = true
w = zeros(d,1)
lambda_i = lambda/n # Regularization for individual example in expectation
gradRecord = zeros(d)
D = zeros((d,d))

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

    # 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
    gradRecord += g_i.^2

    for j in 1:d
        D[j,j] = 1/sqrt(delta + gradRecord[j])
    end


    # Choose the step-size
    alpha = 1/(k^(1/2)) # 1/(lambda_i*k)

    # Take thes stochastic gradient step
    global 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,delta);
        end
        if delta < progTol
            @printf("Parameters changed by less than progTol on pass\n");
            break;
        end
        global w_old = copy(w);
    end
end
```

4. Using the SAG-algorithm we got very good performance with a function value of $f = 2.7068 \cdot 10^4$. The code used is the following

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

# Add bias variable, initialize w, set regularization and optimization parameters
(n,d) = size(X)
lambda = 1

# Initialize
maxPasses = 10
progTol = 1e-4
verbose = true
w = zeros(d,1)
lambda_i = lambda/n # Regularization for individual example in expectation

# Memory for gradients
g = zeros(d)  # effective gradient
```

```
V = zeros((d,n))  # matrix containing memory of v_i

# Computing Lipschitz-constant
(n, d) = size(X)
norms = [norm(X[i,:], 2)^2 for i in 1:n]
L = 0.25*maximum(norms) + lambda_i

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

    # 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

    g = g - V[:, i] + g_i
    V[:, i] = g_i

    # Choose the step-size
    alpha =  1/L

    # Take thes stochastic gradient step
    global w -= alpha*g/n

    # 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,delta);
        end
        if delta < progTol
            @printf("Parameters changed by less than progTol on pass\n");
            break;
        end
        global w_old = copy(w);
    end
end
```

3 Very-Short Answer Questions

1. $f(w) \geq f^*$ for all w: Satisfied by B, C^+ and SC.
2. There exists a stationary point: Satisfied SC.
3. There exists at most one stationary point: Satisfied by C^+ and SC.

4. All stationary points are global optima: Satisfied by C, C^+ and SC.
5. The no free lunch theorem is not violated because the data sets on Kaggle are not representative for all possible data sets, and more advanced algorithms may perform better on these sets.
6. If a function satisfies the Polyak- Lojasiewicz inequality its convergence rate using gradient descent will be linear.
7. Consider the objective function $f(w) = |w|$ with subgradient $g(w) = aw, \forall a \in [-1, 1]$ at its minima $w = 0$, for which the f will increase regardless of step size.
8. L1-regularization on a group is precisely the same as normal L1-regularization, as we can write $\|w\|_{1,1} = \sum |w| = \|w\|_1$, and thus we do not achieve group sparsity.
9. Inductive learning is supervised learning in the traditional sense: we train the model on labeled samples, and our goal is to create a model with the best possible performance on all unseen data. In transductive learning we relax our ambitions, and are instead only interested in predicting the labels on a given test set.
10. The expected cost of one partial derivative is $O(|E|/d)$, where E is the set of edges in the graph, making the algorithm inefficient if the graph is large with many edges.
11. For smooth problems, stochastic subgradient will have lower performance than a deterministic approach because of the randomness involved.
12. Computation time of gradients increase more than the larger step size can compensate.
13. The SVRG reduces variance and memory usage by occasionally computing the exact gradient and thus also leads to faster convergence.