CPSC 540 Assignment 1 (due January 16)

Fundamentals, Convex Functions, Numerical Optimization

- You can work in groups of 1-3 people on the assignments, but please only hand in *one* assignment.
- Place the names and student numbers of all group members on the first page, and submit all answers as a single PDF file to handin.
- Organize your PDF file sequentially according to the section numbers in this document. Place answers (including code/figures) in the appropriate section.
- You will lose marks if answers are unclear or if the TA can't easily find the answers.
- At the start of each question, acknowledge sources of help from outside of class/textbook (classmates outside the group, online material, papers, etc.).
- All Sections are equally weighted, except for Section 0.
- The code and data referred to in the assignment is available in *a1.zip*.
- Any modifications/updates/clarifications after the assignment is first put online will be marked in red.
- Some of the questions require a bit of work, but no individual question is intended to take a lot of time. If you are really stuck, try coming up with a good question and posting on Piazza (other people are likely stuck on the same issue).

0 Course Prerequisite Form

Graduate students in CPSC or EECE must submit the prerequisite form:
`https://www.cs.ubc.ca/~schmidtm/Courses/540_prereqs.pdf`
Hand this in at the start of one of the first 3 lectures, or at the start of the first tutorial on January 12th. Students who don't submit the form may be dropped from the course.

1 Fundamentals

The purpose of this question is to give you practice using the mathematical and coding notation that we will adopt in the course.

1.1 Matrix Notation

For this question we'll use the following Householder-like notation:

1. α is a scalar.
2. w, a, and b are d by 1 column-vectors.

3. y and v are n by 1 column-vectors (with elements y^i and v_i).
4. A is a d by d matrix, not necessarily symmetric (with elements a_{ij}).
5. V is a diagonal matrix with v along the diagonal.
6. B is a diagonal matrix with b along the diagonal.
7. X is a n by d matrix (with rows $(x^i)^T$).

Express the gradient $\nabla f(w)$ and Hessian $\nabla^2 f(w)$ of the following functions in matrix notation, simplifying as much as possible.

1. The linear function
$$f(w) = w^T a + \alpha + \sum_{j=1}^{d} w_j a_j.$$
2. The linear function
$$f(w) = a^T w + a^T A w + w^T A^T b.$$
3. The quadratic function
$$f(w) = w^T w + w^T X^T X w + \sum_{i=1}^{d}\sum_{j=1}^{d} w_i w_j a_{ij}.$$
4. L2-regularized weighted least squares,
$$f(w) = \frac{1}{2}\sum_{i=1}^{n} v_i (w^T x^i - y^i)^2 + \frac{\lambda}{2}\|w\|^2.$$
5. Weighted L2-regularized probit regression,
$$f(w) = -\sum_{i=1}^{n} \log p(y^i | x^i w) + \frac{1}{2}\sum_{j=1}^{d} b_j w_j^2.$$
where $y^i \in \{-1, +1\}$ and the likelihood of a single example i is given by
$$p(y^i | x^i, w) = \Phi(y^i w^T x^i).$$
where Φ is the cumulative distribution function (CDF) of the standard normal distribution.

Hint: You can use the results we showed in class to simplify the derivations. You can use 0 to represent the zero vector or a matrix of zeroes and I to denote the identity matrix. It will help to convert the fourth example to matrix notation first. For the fifth example, it is useful to define a vector c containing the CDF $\Phi(y^i w^T x^i)$ as element c_i and a vector p containing the corresponding PDF as element p_i. For the fifth one you'll need to define new vectors to express the gradient and Hessian in matrix notation (and remember the relationship between the PDF and CDF). As a sanity check, make sure that your results have the right dimension.

Answer:

1.
$$\nabla f(w) = a + 0 + a = 2a.$$
$$\nabla^2 f(w) = 0.$$

2.
$$\nabla f(w) = a + A^T a + A^T b = a + A^T(a + b),$$
or you could alternately factor out the a.
$$\nabla^2 f(w) = 0.$$

3.
$$\nabla f(w) = 2w + 2X^T Xw + (A + A^T)w.$$
$$\nabla^2 f(w) = 2I + 2X^T X + A + A^T.$$

4.
$$f(w) = \frac{1}{2}(Xw - y)^T V(Xw - y) + \frac{\lambda}{2}\|w\|^2.$$
$$\nabla f(w) = X^T V(Xw - y) + \lambda w$$
$$\nabla^2 f(w) = X^T VX + \lambda I.$$

5. Define the vector r with elements
$$r_i = \frac{p_i}{c_i}$$
which is y^i times the derivative of the log(CDF) with respect to its input. We then have
$$\nabla f(w) = Bw - X^T(y \circ r).$$
(There are many variations like defining $r_i = -y_i \frac{p_i}{c_i}$ which would give $\nabla f(w) = X^T r$.) Let's apply the quotient rule applied to r_i to define a diagonal matrix D with elements
$$D_{ii} = -\frac{p_i'}{c_i} + \frac{p_i^2}{c_i^2},$$
where the derivative of the standard normal PDF would be
$$p_i' = -y^i w^T x^i p_i.$$
(I've actually taken the negative of the derivative, since I want the element D_{ii} to be positive.) This lets us write the Hessian as
$$\nabla^2 f(w) = B + X^T DX,$$

1.2 Regularization and Cross-Validation

Download *a1.zip* from the course webpage, and start Matlab in a directory containing the extracted files. If you run the script *example_nonLinear*, it will:

1. Load a one-dimensional regression dataset.
2. Fit a least-squares linear regression model.
3. Report the test error.
4. Draw a figure showing the training/testing data and what the model looks like.

Unfortunately, this is not a great model of the data, and the figure shows that a linear model is probably not suitable.

1. Write a function called *leastSquaresRBFL2* that implements *least squares using Gaussian radial basis functions (RBFs) and L2-regularization.*
   You should start from the *leastSquares* function and use the same conventions: n refers to the number of training examples, d refers to the number of features, X refers to the data matrix, y refers to the targets, Z refers to the data matrix after the change of basis, and so on. Note that you'll have to add two additional input arguments (λ for the regularization parameter and σ for the Gaussian RBF variance) compared to the *leastSquares* function. To make your code easier to understand/debug, you may want to define a new function *rbfBasis* which computes the Gaussian RBFs for a given training set, testing set, and σ value. Hand in your function and the plot generated with $\lambda = 1$ and $\sigma = 1$.

2. When dealing with larger datasets, an important issue is the dependence of the computational cost on the number of training examples n and the number of features d. What is the cost in big-O notation of training the model on n training examples with d features under (a) the linear basis, and (b) Gaussian RBFs (for a fixed σ)? What is the cost of classifying t new examples under these two bases? Assume that multiplication by an n by d matrix costs $O(nd)$ and that inverting a d by d linear system costs $O(d^3)$.

3. Modify the training/validation procedure to use 10-fold cross-validation on the training set to select λ and σ. Hand in your cross-validation procedure and the plot you obtain with the best values of λ and σ

Note: If you find that calculating the Euclidean distances between all pairs of points takes too long, the following code will form a matrix containing the squared Euclidean distances between all training and test points:

```
[n,d] = size(X);
[t,d] = size(Xtest);
D = X.^2*ones(d,t) + ones(n,d)*(Xtest').^2 - 2*X*Xtest';
```

Element $D(i, j)$ gives the squared Euclidean distance between training point i and testing point j.

Answer:

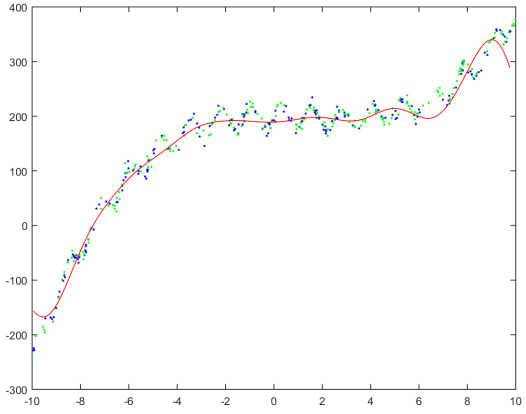
1.

The code should look like this:

```
function [model] = leastSquaresRBFL2(X,y,lambda,sigma)

% Compute sizes
[n,d] = size(X);

% From Gaussian RBFs
Z = rbfBasis(X,X,sigma);
d = size(Z,2);

% Solve least squares problem
w = (Z'*Z + lambda*eye(d))\Z'*y;

model.X = X;
model.w = w;
model.sigma = sigma;
model.predict = @predict;

end
```

```
function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);

Zhat = rbfBasis(Xhat,model.X,model.sigma);

yhat = Zhat*model.w;
end

function [Xrbf] = rbfBasis(X1,X2,sigma)
n1 = size(X1,1);
n2 = size(X2,1);
d = size(X1,2);
Z = 1/sqrt(2*pi*sigma^2);
D = X1.^2*ones(d,n2) + ones(n1,d)*(X2').^2 - 2*X1*X2';
Xrbf = Z*exp(-D/(2*sigma^2));
end
```

The plot should look like this:



2.

Under the linear basis, the dominant costs are: computing $X^T X$ which costs $O(nd^2)$ to compute $O(d^2)$ inner products at a cost of $O(n)$ each, and inverting $X^T X$ which costs $O(d^3)$ using standard methods. This gives a cost of $O(nd^2 + d^3)$. Classifying t new examples costs $O(td)$ to compute t inner products at cost of $O(d)$ each.

Using RBFs, forming X_{rbf} costs $O(n^2 d)$ to compute $O(n^2)$ distances at a cost of $O(d)$ each. Computing $X_{\text{rbf}}^T X_{\text{rbf}}$ costs $O(n^3)$ while inverting it also costs $O(n^3)$ using standard methods. This gives a cost of $O(n^2 d + n^3)$. Classifying a new example costs $O(tnd)$.

RBFs are cheaper to train with $n < d$. RBFs are never cheaper to test.

3.

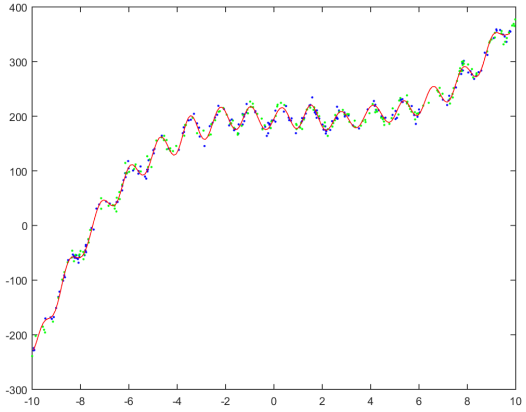
The code should look roughly like this:

```
% Clear variables and close figures
clear all
close all
```

```
% Load data
load nonLinear.mat % Loads {X,y,Xtest,ytest}
[n,d] = size(X);
[t,~] = size(Xtest);

% Find best value of RBF kernel parameter,
%   training on 9/10 of the train set and validating on the remaining
minErr = inf;
nSplits = 5;
for sigma = 2.^[-15:15]
    for lambda = 2.^[-15:15]

        validError = 0;
        for split = 1:nSplits

            % Get the training set and test set indices
            testStart = 1 + (n/nSplits)*(split-1);
            testEnd = (n/nSplits)*split;
            trainNdx = [1:testStart-1 testEnd+1:n];
            testNdx = testStart:testEnd;

            % Train on the training set
            model = leastSquaresRBFL2(X(trainNdx,:),y(trainNdx),lambda,sigma);

            % Compute the error on the validation set
            yhat = model.predict(model,X(testNdx));
            validError = validError + sum((yhat - y(testNdx)).^2)/(n/nSplits);
        end
        fprintf('Error with lambda = %.3e, sigma = %.3e = %.2f\n',lambda,sigma,validError)

        % Keep track of the lowest validation error
        if validError < minErr
            minErr = validError;
            bestLambda = lambda;
            bestSigma = sigma;
        end
    end
end
fprintf('Value of lambda and sigma that achieved the lowest validation error were %.3e and

% Train least squares model on training data
model = leastSquaresRBFL2(X,y,bestLambda,bestSigma);

% Test least squares model on test data
yhat = model.predict(model,Xtest);

% Report test error
squaredTestError = sum((yhat-ytest).^2)/t

% Plot model
```

```
figure(1);
plot(X,y,'b.');
hold on
plot(Xtest,ytest,'g.');
Xhat = [min(X):.1:max(X)]'; % Choose points to evaluate the function
yhat = model.predict(model,Xhat);
plot(Xhat,yhat,'r');
ylim([-300 400]);
print -dpng nonLinear2.png
```

The plot should look like this:



The best values seem to be something like $\sigma = \frac{1}{2}$ and $\lambda \approx 10^{-5}$.

1.3 MAP Estimation

In class, we showed that under the assumptions

$$y^i \sim \mathcal{N}(w^T x^i, 1), \quad w_j \sim \mathcal{N}\left(0, \frac{1}{\lambda}\right),$$

the MAP estimate is equivalent to solving the L2-regularized least squares problem

$$f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w\|^2,$$

in the "loss plus regularizer" framework. For each of the alternate assumptions below, write it in the "loss plus regularizer" framework (simplifying as much as possible, including converting to matrix notation):

1. Laplace distribution likelihoods and priors,

$$y^i \sim \mathcal{L}(w^T x^i, 1), \quad w_j \sim \mathcal{L}\left(0, \frac{1}{\lambda}\right).$$

2. Gaussians with separate variance for each training example and variable,

$$y^i \sim \mathcal{N}(w^T x^i, \sigma_i^2), \quad w_j \sim \mathcal{N}\left(0, \frac{1}{\lambda_j}\right).$$

3. Poisson-distributed likelihood (for the case where y^i represents discrete counts) and Gaussian prior,

$$y^i \sim \mathcal{P}(\exp(w^T x^i)), \quad w_j \sim \mathcal{N}\left(0, \frac{1}{\lambda}\right),$$

Answer:

1. $$f(w) = \|Xw - y\|_1 + \lambda\|w\|_1.$$

2. Define Θ as a diagonal matrix with the $1/\sigma_i^2$ along the diagonal and similarly for Λ and the λ_j,

$$f(w) = \frac{1}{2}(Xw - y)^T\Theta(Xw - y) + \frac{1}{2}w^T\Lambda w,$$

or if you use quadratic norms you can get

$$f(w) = \frac{1}{2}\|Xw - y\|_\Theta^2 + \frac{1}{2}\|w\|_\Lambda^2.$$

3. This gives a likelihood of

$$p(y^i|w^T x^i) = \frac{\exp(y^i w^T x^i)\exp(-\exp(w^T x^i))}{y^i!},$$

which leads to an objectve of

$$f(w) = \sum_{i=1}^n \left(-y^i w^T x^i + \exp(w^T x^i)\right) + \frac{\lambda}{2}\|w\|^2$$

If we wanted, we could write define $v_i = \exp(w^T x^i)$ and write this in matrix notation as

$$f(w) = -y^T Xw + 1^T v + \frac{\lambda}{2}\|w\|^2.$$

2 Convex Functions

2.1 Minimizing Strictly-Convex Quadratic Functions

Solve for the minimizer of the below strictly-convex quadratic functions:

1. $f(w) = \frac{1}{2}\|w - v\|^2$ (projection of v onto real space).
2. $f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{1}{2}w^T\Lambda w$ (least squares with quadratic-norm regularization).
3. $f(w) = \frac{1}{2}\sum_{i=1}^n v_i(w^T x_i - y_i)^2 + \frac{\lambda}{2}\|w - w^0\|^2$ (weighted least squares shrunk towards non-zero w^0).

Above we assume that v and w^0 are d by 1 vectors, and Λ is a d by d positive-definite matrix. You can use V as a diagonal matrix containing the v_i values along the diagonal.

Answer:

1. $f(w) = \frac{1}{2}\|w\|^2 - w^T v + \frac{1}{2}\|v\|^2$ so $\nabla f(w) = w - v$ and setting it equal to 0 we have $w = v$.
2. $\nabla f(w) = X^T Xw - X^T y + \Lambda w$ so solving for w we get $w = (X^T X + \Lambda)^{-1}X^T y$.
3. $f(w) = \frac{1}{2}(Xw - y)^T V(Xw - y) + \frac{\lambda}{2}(w - w^0)^T(w - w^0)$ and $\nabla f(w) = X^T VXw - X^T Vy + \lambda w - \lambda w^0$, so the solution is $w = (X^T VX + \lambda I)^{-1}(X^T Vy + \lambda w^0)$.

2.2 Proving Convexity

Show that the following functions are convex using one of the definitions of convexity (without using the "operations that preserve convexity" or using convexity results stated in class):

1. Negative logarithm	$f(w) = -\log(aw)$	$w > 0$
2. Quadratic with positive semi-definite A	$f(w) = \frac{1}{2}w^T Aw + b^T w + \gamma$	$w \in \mathbb{R}^d, A \succeq 0$
3. Any norm	$f(w) = \|w\|_p$	$w \in \mathbb{R}^d$
4. Logistic regression	$f(w) = \sum_{i=1}^n \log(1 + \exp(-y_i w^T x_i))$	$w \in \mathbb{R}^d$

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

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

5. $f(w) = \|Xw - y\|_p + \lambda\|Aw\|_q$	(regularized regression with arbitrary p-norm and weighted q-norm)
6. $f(w) = \sum_{i=1}^N \max\{0, \|w^T x_i - y_i\| - \epsilon\} + \frac{\lambda}{2}\|w\|_2^2$	(support vector regression)
7. $f(w) = \max_{\{ijk \mid i \neq j, i \neq k, j \neq k\}}\{\|w_i\| + \|w_j\| + \|w_k\|\}$	(3 largest-magnitude elements)

Answer:

1.

$$f'(w) = -\frac{1}{w},$$

$$f''(w) = \frac{1}{w^2},$$

which implies convexity because $w > 0$.

2.

$$\nabla f^2(x) = A,$$

which implies convexity because $A \succeq 0$.

3. Using the triangle inequality followed by absolute homogeneity and $0 \leq \theta \leq 1$ we have

$$\|\theta x + (1-\theta)y\|_p \leq \|\theta x\| + \|(1-\theta)y\|_p = |\theta|\|x\|_p + |1-\theta|\|y\|_p = \theta\|x\|_p + (1-\theta)\|y\|_p,$$

which is the zero-order definition of convexity.

4. In class we showed that the Hessian has the form

$$\nabla^2 f(w) = X^T D X,$$

where the matrix D is diagonal with elements given by the product of sigmoid functions. Since sigmoid functions are non-negative, we can take the square root of D and write any quadratic form as

$$v^T X^T D^{1/2} D^{1/2} X v = \|X D^{1/2} v\|^2 \geq 0.$$

5. We showed above that $\|w\|_q$ is a convex function above. The function $\|Xw - y\|_p$ is convex because we are composing the convex $\|\cdot\|_p$ with an affine function $Xw - y$. Finally, f is convex since $\lambda > 0$ so we are taking a non-negative weighted combination.

6. First, we note that $|w^T x_i - y|$ is a norm composed with an affine function so it is convex (you could also re-write the absolute value as the maximum of two linear functions). We then have that $\max\{0, |w^T x_i - y_i| - \epsilon\}$ is a max of convex functions so it is convex. We have that $\|w\|^2$ is convex since its Hessian is the identity

matrix (note that this is not a norm because it is squared). Finally, since $\lambda > 0$ we have that f is a non-negative sum of convex terms so it is convex.

7. First, we note that the absolute value functions $|x_i|$ are convex because they are norms. The sum of the absolute values is also convex because sums preserve convexity. Thus, given any choice of three variables we have a convex function. Thus, if we take the maximum over *sets of 3* variables then it's a maximum over convex functions and is convex.

Alternately, you could show that the maximum over absolute values of sets of 3 is a norm, so it's convex because all norms are convex.

2.3 Robust Regression

The script *example_outliers* loads a one-dimensional regression dataset that has a non-trivial number of 'outlier' data points. These points do not fit the general trend of the rest of the data, and pull the least squares model away from the main cluster of points. One way to improve the performance in this setting is simply to remove or downweight the outliers. However, in high-dimensions it may be difficult to determine whether points are indeed outliers (or the errors might simply be heavy-tailed). In such cases, it is preferable to replace the squared error with an error that is more robust to outliers.

1. Write a new function, *robustRegression(X,y)*, that adds a bias variable and fits a linear regression model by minimizing the absolute error instead of the square error,

$$f(w) = \|Xw - y\|_1.$$

   You should turn this into a *linear program* as shown in class, and you can solve this linear program using Matlab's *linprog* function. Hand in the new function and report the minimum absolute training error that is possible on this dataset.

2. There have been several attempts to adapt SVMs to the regression problem. The most common method for support vector regression uses what iscalled the ϵ-insensitive loss,

$$f(w) = \sum_{i=1}^{n} \max\{0, |w^T x^i - y^i| - \epsilon\}.$$

   Here, ϵ is a parameter and notice that the model only penalizes errors larger than ϵ. Show how to write minimizing this objective function as a linear program.

3. Write a new function, *svRegression(X,y,epsilon)*, that minimizes the ϵ-insensitive objective. Hand in the new function and report the absolute training error achieved with $\epsilon = 1$..

Answer:

1.

The lowest possible average absolute error is 3.0666 (or 3.0755 without the bias variable), which is lower than the 4.0068 of the least squares model. One possible implementation would be

```
function [model] = robustRegression(X,y)

[n,d] = size(X);

Z = [ones(n,1) X];
d = d+1;

% Cost vector for linear program:
```

```
% - the first d values are for variables associated with w
% - the last n values are for the upper bounds on the absolute values
c = [zeros(d,1);ones(n,1)];

% Constraints will be w'x^i - v_i <= y_i and -w'x^i - v_i <= -y_i
A = [Z -eye(n);-Z -eye(n)];
b = [y;-y];

[wv] = linprog(c,A,b);

w = wv(1:d);
model.w = w;
model.predict = @predict;

end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);
Zhat = [ones(t,1) Xhat];
w = model.w;
yhat = Zhat*w;
end
```

2.

Let's get rid of the absolute value by taking a maximum over 3 linear terms,

$$f(w) = \sum_{i=1}^{n} \max\{0, w^T x^i - y^i - \epsilon, y^i - w^T x^i - \epsilon\},$$

Now introducing slack variables we get

$$f(w) = \sum_{i=1}^{n} v_i.$$

with the constraints

$$v_i \geq 0, \quad v_i \geq w^T x^i - y^i - \epsilon, \quad v_i \geq y^i - w^T x^i - \epsilon,$$

which is a linear objective with linear constraints.

3.

The code could look like this:

```
function [model] = svRegression(X,y,epsilon)

[n,d] = size(X);

Z = [ones(n,1) X];
d = d+1;

% Linear term
c = [zeros(d,1);ones(n,1)];

% Constraints will be w'x^i - v_i <= y_i + epsilon and -w'x^i - v_i <= -y_i + epsilon
```

```
A = [Z -eye(n);-Z -eye(n)];
b = [y+epsilon;-y+epsilon];

LB = [-inf(d,1);zeros(n,1)]; % v_i >= 0

[wv] = linprog(c,A,b,[],[],LB,[]);

find(wv(d+1:end) > 1)
w = wv(1:d);
model.w = w;
model.predict = @predict;

end

function [yhat] = predict(model,Xhat)
[t,d] = size(Xhat);
Zhat = [ones(t,1) Xhat];
w = model.w;
yhat = Zhat*w;
end
```

The average absolute error with this approach is 3.0746.

3 Numerical Optimization

3.1 Gradient Descent and Newton's Method

The function *example_gradient* loads a simple binary classification dataset, and fits an ℓ_2-regularized logistic regression model to it using the *findMin* function which implements a simple gradient descent algorithm. The *findMin* function is generic in the sense that it only needs an anonymous function which computes the objective value and gradient given a parameter vector. On each iteration *findMin* uses a backtracking line-search to find a step-size α that satisfies the Armijo "sufficient decrease" condition. It always tries $\alpha = 1$ first, and whenever the condition is not satisfied it uses "cubic Hermite polynomial" interpolation to find a smaller value of α to try. It continues running the algorithm until a pre-specified number of iterations are reached or until the norm of the gradient is sufficiently small. On this dataset, the method only requires 9 iterations before it satisfies its optimality condition, although it backtracks 13 times and evaluates the function and gradient a total of 23 times.

Report the effect on performance (in terms of number of backtracking iterations and total number of iterations) of making the following changes to *findMin*:

1. When backtracking, replacing the cubic-Hermite interpolation with the simpler strategy of dividing α in half (as suggested in the Boyd & Vandenberghe book).
2. Instead of resetting α to one after the line-search, set it using the Barzilai-Borwein step-size, given by

$$\alpha \leftarrow -\alpha \frac{v^T \nabla f(w)}{v^T v},$$

where v is the new gradient value minus the old gradient value.

3. Fix the step-size α to $1/L$, where L is given by

$$L = \frac{1}{4}\max\{\text{eig}(X^T X)\} + \lambda,$$

which is the Lipschitz constant of the gradient.

4. Instead of using the gradient direction, set d to the Newton direction which is given by

$$d = [\nabla^2 f(w)]^{-1}\nabla f(w).$$

For the Newton direction, you'll need to make a new objective function that returns the Hessian in addition to the function and gradient, and modify *findMin* to use the Hessian.

Answer:

1. If you use step-size halving, the method backtracks at least 4 times on every iteration and the total number of evaluations increases to 83.
2. With the Barzilai-Borwein step-size, the method only backtracks once (after the first iteration) and the total number of evaluations decreases to 17.
3. With the Lipschitz constant, the algorithm never backtracks but the total number of evaluations increases to 34.
4. With the Newton direction, the algorithm never backtracks and only 5 iterations are needed.

A variation of findMin that includes an extra option *optimOption* to implement any of these changes is below:

```
function [w,f] = findMin(funObj,w,maxEvals,verbose,optimOption,varargin)
% Find local minimizer of differentiable function

% Parameters of the Optimizaton
optTol = 1e-2;
gamma = 1e-4;

% Evaluate the initial function value and gradient
if optimOption == 4
    [f,g,H] = funObj(w,varargin{:});
else
    [f,g] = funObj(w,varargin{:});
end
funEvals = 1;

alpha = 1;
while 1
    %% Compute search direction
    if optimOption == 4
        d = H\g;
    else
        d = g;
    end

    if optimOption == 3
        alpha = 1/130.7460;
```

```
end

%% Line-search to find an acceptable value of alpha
    w_new = w - alpha*d;
if optimOption == 4
    [f_new,g_new,H_new] = funObj(w_new,varargin{:});
else
    [f_new,g_new] = funObj(w_new,varargin{:});
end
    funEvals = funEvals+1;

dirDeriv = g'*d;
while f_new > f - gamma*alpha*dirDeriv
    if verbose
        fprintf('Backtracking...\n');
    end
    if optimOption == 1
        alpha = alpha/2;
    else
        alpha = alpha^2*dirDeriv/(2*(f_new - f + alpha*dirDeriv));
    end
    w_new = w - alpha*d;
    if optimOption == 4
        [f_new,g_new,H_new] = funObj(w_new,varargin{:});
    else
        [f_new,g_new] = funObj(w_new,varargin{:});
    end
    funEvals = funEvals+1;
end
alphaFinal = alpha;

%% Update step-size for next iteration
if optimOption == 2
    y = g_new - g;
    alpha = -alpha*(y'*g)/(y'*y);
else
    alpha = 1;
end
%% Sanity check on step-size
if ~isLegal(alpha) || alpha < 1e-10 || alpha > 1e10
   alpha = 1;
end

%% Update parameters/function/gradient
w = w_new;
f = f_new;
g = g_new;
if optimOption == 4
    H = H_new;
end
```

```
    %% Test termination conditions
        optCond = norm(g,'inf');
    if verbose
        fprintf('%6d %15.5e %15.5e %15.5e\n',funEvals,alphaFinal,f,optCond);
    end

        if optCond < optTol
        if verbose
            fprintf('Problem solved up to optimality tolerance\n');
        end
                break;
        end

        if funEvals >= maxEvals
        if verbose
            fprintf('At maximum number of function evaluations\n');
        end
                break;
        end
end
end

function [legal] = isLegal(v)
legal = sum(any(imag(v(:))))==0 & sum(isnan(v(:)))==0 & sum(isinf(v(:)))==0;
end
```

3.2 Hessian-Free Newton

The dataset used in the previous question leads to an easy optimization problem. If you apply the variations in the previous question to the larger dataset contained in *rcv1_train_binary.mat* then the performance is very different:

1. Without modifications, *findMin* doesn't work since the objective overflows.[1]
2. Step-size halving avoids the overflow, but the method doesn't reaches a reasonable accuracy even after 500 iterations.
3. With the Barzilai-Borwein step-size, it reaches a solution in a reasonable number of iterations.
4. Computing L is slower than solving the original problem because d is so large.[2]
5. Computing the Hessian is slower than solving the original problem because d is so large.

In Hessian-free Newton methods, we compute an approximate Newton direction d without ever forming the Hessian. The standard way to do this is to use *conjugate gradient* to solve for d, which only requires Hessian-vector products of the form $\nabla^2 f(w)v$ for a vector v. Note that Hessian-vector products can always be computed at a similar cost to computing the gradient. For example, for logistic regression we can use

$$\nabla^2 f(w)v = X^T DXv = X^T(D(Xv)),$$

[1] There are two standard solutions to this problem: we could detect the overflow and backtrack out of regions where it overflows, or we could use the *log-sum-exp* trick to evaluate the objective without overflowing.

[2] An often-faster way to compute the largest eigenvalue is the "power method".

and the order of operations leads to a cost of $O(nd)$. This is cheaper than the $O(nd^2)$ cost of forming the Hessian. Use Matlab's *pcg* function to implement a "Hessian-free" Newton's method, where you use conjugate gradient to solve the Newton system. Report the output of *findMin* on *rcv1_train_binary.mat* when using this strategy and using *optTol* as the tolerance for *pcg*.

Hint: In *logisticL2*, define a function that computes the Hessian-vector product and has the following header:

```
function [Hv] = Hvfunc(w,v,X,y,lambda)
```

To define the *AFUN* argument needed by *pcg* (which must be a function of v for a fixed w) you can use an anonymous function:

```
Hv = @(v)Hvfunc(w,v,varargin{:});
```

Answer: The output I got was: pcg converged at iteration 12 to a solution with relative residual 0.0071
2 1.00000e+00 5.18715e+03 6.38540e+01
pcg converged at iteration 9 to a solution with relative residual 0.0089
3 1.00000e+00 4.22805e+03 1.80539e+01
pcg converged at iteration 9 to a solution with relative residual 0.0071
4 1.00000e+00 4.09305e+03 1.02721e+01
pcg converged at iteration 8 to a solution with relative residual 0.0088
5 1.00000e+00 4.08556e+03 2.29591e+00
pcg converged at iteration 8 to a solution with relative residual 0.0053
6 1.00000e+00 4.08547e+03 4.88609e-02
pcg converged at iteration 8 to a solution with relative residual 0.008
7 1.00000e+00 4.08547e+03 7.96543e-05
Problem solved up to optimality tolerance

You could get better performance by basing the tolerance on the norm of the gradient, which is called a "forcing sequence".

3.3 Multi-Class Logistic Regression

The function *example_multiClass* loads a multi-class classification dataset and fits a "one-vs-all" logistic regression classifier, then reports the validation error and shows a plot of the data/classifier. The performance on the validation set is ok, but could be much better. For example, this classifier never even predicts some of the classes.

Using a one-vs-all classifier hurts performance because the classifiers are fit independently, so there is no attempt to calibrate the columns of the matrix W. An alternative to this independent model is to use the softmax probability,

$$p(y^i|W, x^i) = \frac{\exp(w_{y^i}^T x^i)}{\sum_{c=1}^k \exp(w_c^T x^i)}.$$

Here c is a possible label and w_c is column c of W. Similarly, y^i is the training label, w_{y^i} is column y^i of W. The loss function corresponding to the negative logarithm of the softmax probability is given by

$$f(W) = \sum_{i=1}^n \left[-w_{y^i}^T x^i + \log\left(\sum_{c'=1}^k \exp(w_{c'}^T x^i) \right) \right].$$

Make a new function, *softmaxClassifier*, which fits W using the softmax loss from the previous section instead of fitting k independent classifiers. Hand in the code and report the validation error.

Hint: you can use the *derivativeCheck* function to help you debug the gradient of the softmax loss. It can also help you numerically check your answer to several more of this assignment's questions.

Answer:
The code should look something like this:

```
function [model] = softmaxClassifier(X,y)

% Compute sizes
[n,d] = size(X);
k = max(y);

W = zeros(d,k); % Each column is a classifier
W(:) = findMin(@softmaxLoss,W(:),500,1,X,y,k);

model.W = W;
model.predict = @predict;
end

function [yhat] = predict(model,X)
W = model.W;
[~,yhat] = max(X*W,[],2);
end

function [nll,g,H] = softmaxLoss(w,X,y,k)

[n,p] = size(X);
W = reshape(w,[p k]);

XW = X*W;
Z = sum(exp(XW),2);

ind = sub2ind([n k],[1:n]',y);
nll = -sum(XW(ind)-log(Z));

g = zeros(p,k);
for c = 1:k
    g(:,c) = X'*(exp(XW(:,c))./Z-(y == c));
end
g = reshape(g,[p*k 1]);
end
```

(Longer versions that use a 'for' loop instead of *sub2ind* are ok.) The validation error depends on how precisely you solve the optimization problem, but it decreases to something very small like 0.01 or 0.02.