Econ 753 Assignment 2

Katherine Fairley

Method	Estimated Parameters						Iterations	Function Evaluations	Time
Quasi-Newton w/ BFGS and numerical derivative	2.534	-0.032	0.116	-0.354	0.08	-0.409	54	427	0.0297
Quasi-Newton w/ BFGS and analytical derivative	2.534	-0.032	0.116	-0.354	0.08	-0.409	54	61	0.0138
Nelder-Mead	2.534	-0.032	0.116	-0.354	0.08	-0.409	1109	1853	0.0692
BHHH	2.534	-0.032	0.116	-0.354	0.08	-0.409	104	104	0.0465
NLLS	2.513	-0.038	0.114	-0.280	0.068	-0.369	13	13	0.0036

The eigenvalues for the initial Hessian approximation from the BHHH method are:

$$\{0.0002, 0.0087, 0.0094, 0.0229, 0.0828, 9.5972\}$$

The eigenvalues for the Hessian approximation at the estimated parameters are:

$$\{0.0002, 0.0070, 0.0087, 0.0197, 0.0706, 8.1680\}$$

```matlab

% Assignment 2 Econ 753
% Set working directory
cd('C:\Users\KatyK\University of Michigan Dropbox\Katherine Fairley\Notability\Spring 2025\753 - Methods\Github')

% Load and prepare data
data = csvread('psychtoday.csv', 1);  % Skip header row
y = data(:,1);                        % Dependent variable
X = data(:,2:end);                    % Independent variables
beta_init = zeros(6,1);               % Initial parameter values

%% Question 1.1: Quasi-Newton with BFGS and numerical derivative
[beta_numeric, output_numeric, time_numeric] = maximize_numgrad(X, y, beta_init);

%% Question 1.2: Quasi-Newton with BFGS and analytical derivative
[beta_analytic, output_analytic, time_analytic] = maximize_analyticgrad(X, y, beta_init);

%% Question 1.3: Nelder-Mead Optimization
% Define objective function (negative log-likelihood for minimization)
obj_fun = @(beta) -loglikelihood(X, y, beta);
options = optimset('TolX', 1e-12, 'MaxFunEvals', 1e4);

tic;  % Start timer
[beta_nm, fval, exitflag, output] = fminsearch(obj_fun, beta_init, options);
time_nm = toc;
output_nm = output;

%% Question 1.4: BHHH Algorithm
[beta_bhhh, convergence_flag] = bhhh_poisson(X, y, beta_init);

%% Question 3: NLLS Algorithm
% Initialize parameters
beta = beta_init;
n_obs = length(y);
n_params = length(beta_init);
tolerance = 1e-12;
iter_count = 0;
func_evals = 0;
tic;

% Initialize residual for convergence check
```

```matlab
residual_old = Inf;
converged = false;

while ~converged
    iter_count = iter_count + 1;

    % Calculate predicted values and residuals
    pred_values = exp(X * beta);
    residuals = y - pred_values;
    func_evals = func_evals + 1;

    % Check convergence using relative residual change
    rel_residual_change = abs(norm(residuals) - norm(residual_old)) /
       norm(residual_old);
    if rel_residual_change < tolerance && iter_count > 1
        converged = true;
        break;
    end
    residual_old = residuals;

    % Calculate Jacobian matrix
    jacobian = X .* pred_values;

    % Calculate approximate Hessian
    hessian = jacobian' * jacobian;

    % Add regularization if Hessian is near-singular
    if rcond(hessian) < 1e-12
        hessian = hessian + 1e-4 * eye(n_params);
    end

    % Calculate parameter update
    delta = hessian \ (jacobian' * residuals);
    beta_new = beta + delta;
    beta = beta_new;
end

time_nlls = toc;

%% Helper Functions

function [beta_opt, output, time] = maximize_numgrad(X, y, beta_init)
    % Maximizes log-likelihood using BFGS with numerical gradient
    tic;

    % Set optimization options
    options = optimoptions('fminunc', ...
```

```
        'SpecifyObjectiveGradient', false, ...    % Use numerical derivatives
        'Algorithm', 'quasi-newton', ...
        'Display', 'iter', ...
        'StepTolerance', 1e-12);

    % Define objective function (negative log-likelihood)
    obj_fun = @(beta) -loglikelihood(X, y, beta);

    % Minimize negative log-likelihood
    [beta_opt, fval, exitflag, output] = fminunc(obj_fun, beta_init, options);
    time = toc;
end

function ll = loglikelihood(X, y, beta)
    % Computes Poisson log-likelihood
    % Input:
    %   X: matrix of independent variables
    %   y: vector of dependent variable
    %   beta: parameter vector
    % Output:
    %   ll: log-likelihood value
    ll = (-exp(X*beta) + y.*(X*beta) - log(factorial(y))).' * ones(height(y),1);
end

function [beta_opt, output, time] = maximize_analyticgrad(X, y, beta_init)
    % Maximizes log-likelihood using BFGS with analytical gradient
    tic;

    options = optimoptions('fminunc', ...
        'SpecifyObjectiveGradient', true, ...     % Use analytical gradient
        'Algorithm', 'quasi-newton', ...
        'Display', 'iter', ...
        'StepTolerance', 1e-12);

    obj_fun = @(beta) loglike_with_grad(X, y, beta);
    [beta_opt, fval, exitflag, output] = fminunc(obj_fun, beta_init, options);
    time = toc;
end

function [f, g] = loglike_with_grad(X, y, beta)
    % Computes log-likelihood and its gradient
```

```
    % Input:
    %   X: matrix of independent variables
    %   y: vector of dependent variable
    %   beta: parameter vector
    % Output:
    %   f: negative log-likelihood value
    %   g: gradient vector

    X_beta = X * beta;
    exp_X_beta = exp(X_beta);
    f = -(-sum(exp_X_beta) + y'*X_beta - sum(log(factorial(y))));
    g = -(X' * (y - exp_X_beta));
end

function score = score_function(beta, X, y)
    % Computes score (gradient of log-likelihood)
    score = X' * (y - exp(X*beta));
end

function [beta, converged] = bhhh_poisson(X, y, beta_init)
    % Implements BHHH algorithm for Poisson regression
    % Input:
    %   X: matrix of independent variables
    %   y: vector of dependent variable
    %   beta_init: initial parameter values
    % Output:
    %   beta: estimated parameters
    %   converged: convergence flag

    tolerance = 1e-6;
    beta = beta_init;
    n_obs = length(y);

    iter_count = 0;
    func_evals = 0;
    tic;

    % Calculate initial Hessian approximation and eigenvalues
    mu_init = exp(X*beta_init);
    scores_init = zeros(n_obs, length(beta_init));
    for i = 1:n_obs
        scores_init(i,:) = X(i,:)' * (y(i) - mu_init(i));
    end
    hessian_init = scores_init' * scores_init;
    eig_init = eig(hessian_init);

    beta_new = beta + 2*tolerance;
```

```

    while norm(beta_new - beta) >= tolerance
        iter_count = iter_count + 1;
        beta = beta_new;

        % Calculate predicted values and log-likelihood
        mu = exp(X*beta);
        ll = (-mu + y.*X*beta - log(factorial(y))).' * ones(height(y)
           ,1);
        func_evals = func_evals + 1;

        % Calculate scores for each observation
        scores = zeros(n_obs, length(beta));
        for i = 1:n_obs
            scores(i,:) = X(i,:)' * (y(i) - mu(i));
        end

        % Calculate BHHH Hessian approximation
        hessian = scores' * scores;
        total_score = sum(scores, 1)';

        % Add regularization if Hessian is near-singular
        if rcond(hessian) < 1e-12
            hessian = hessian + 1e-6 * eye(size(hessian));
        end

        % Update parameters
        delta = hessian \ total_score;
        beta_new = beta + delta;
    end

    % Calculate final Hessian approximation and eigenvalues
    mu_final = exp(X*beta_new);
    scores_final = zeros(n_obs, length(beta));
    for i = 1:n_obs
        scores_final(i,:) = X(i,:)' * (y(i) - mu_final(i));
    end

    hessian_final = scores_final' * scores_final;
    eig_final = eig(hessian_final);

    % Print summary statistics
    elapsed_time = toc;
    fprintf('Converged in %d iterations\n', iter_count);
    fprintf('Function evaluations: %d\n', func_evals);
    fprintf('Elapsed time: %.4f seconds\n', elapsed_time);
    fprintf('\nEigenvalues of initial Hessian approximation:\n');
```

```
    disp(eig_init);
    fprintf('\nEigenvalues of final Hessian approximation:\n');
    disp(eig_final);
    converged = true;
end
```