# CPSC 540: Assignment 2

Yiwei Hou(84435156)
Xiaomeng Ju(86475150)
Tingting Yu(74439118)

# 1 Convergence Rates

## 1.1 Gradient Descent

1. If $f$ is a $C^1$ function, strong convexity means $f(y) \geq f(x) + < f'(y), y - x > + \frac{u}{2}||x - y||^2$ for $u > 0$ and $y, x$ in its domain. Applying this inequality at $x^t$ and $x^*$, we have $f(x^t) \geq f(x^*) + < f'(x^*), x^* - x^T > + \frac{u}{2}||x^* - x^t||^2$. Since $f'(x^*) = 0$, $\frac{u}{2}||x^* - x^t||^2 \leq f(x^t) - f(x^*) = O(\rho^t)$. This means the iteration $||x^t - x^*||$ has a convergence rate of $O(\rho^{t/2})$.

   If $f$ is a $C^2$ function, by Taylor's theorem, there is a $z$ such that $f(x^t) = f(x^*) + \nabla f(x^*)(x^t - x^*) + \frac{1}{2}(x^t - x^*)^T \nabla^2 f(u)(x^t - x^*)$. By strong convexity, $f(x^*) \geq f(x^t) + \nabla f(x^*)(x^t - x^*) + \frac{u}{2}||x^t - x^*||^2$. Again, as $\nabla f(x^*) = 0$, we have $\frac{u}{2}||x^* - x^t||^2 \leq f(x^t) - f(x^*) = O(\rho^t)$, which means the iteration has a convergence rate of $O(\rho^{t/2})$.

2. From the descent lemma, we know $f(x^{t+1}) \leq f(x^t) + \nabla f(x^t)^T(x^{t+1} - x^t) + \frac{L}{2}||x^{t+1} - x^t||^2$. With the constant step size, $x^{t+1} - x^t = \alpha \nabla f(x^t)$. Plugging this into the descent lemma, we get

$$\begin{aligned} f(x^{t+1}) &\leq f(x^t) - \alpha \nabla f(x^t)^T \nabla f(x^t) + \frac{L\alpha^2}{2}||\nabla f(x^t)||^2 \\ &= f(x^t) - (\alpha - \frac{L\alpha^2}{2})||\nabla f(x^t)||^2 \end{aligned}$$

   Comparing the upper bound with the proof in class, we define $\frac{1}{2\tilde{L}} = \alpha - \frac{L\alpha^2}{2}$ and only need to show $\tilde{L} \geq L$ in order to preserve the linear convergence rate. Since $\alpha < \frac{L}{2}$, the minimum value of $\tilde{L}$ is $L$. Thus, we show that the convergence rate is linear when using a constant step size.

   3. By descent lemma, we have

$$\begin{aligned} f(x^{t+1}) &\leq f(x^t) - \frac{1}{L^t} \nabla f(x^t)^T \nabla f(x^t) + \frac{L}{2L^{t2}}||\nabla f(x^t)||^2 \\ &= f(x^t) - (\frac{1}{L^t} - \frac{L}{2L^{t2}})||\nabla f(x^t)||^2 \end{aligned}$$

   To satisfy the inequality, we need $\frac{1}{L^t} - \frac{L}{2L^{t2}} \geq \frac{1}{2L^t}$ which gives us $L^t \geq L$. Since we start with some $L^0$ that is smaller than $L$ and double the value when the inequality is not satisfied, $L^t \leq 2L$ always holds. Following the steps in class, we have $f(x^t) - f(x^*) \leq (1 - \frac{u}{L^0})(1 - \frac{u}{L^1}) \cdots (1 - \frac{1}{L^{t-1}})(f(x^0) - f(x^*)) \leq (1 - \frac{u}{2L})^t (f(x^0) - f(x^*))$ .

4. When $L^t > L$ for any $t$, the convergence rate would be faster than $\rho = 1 - \frac{u}{L}$.

## 1.2 Sign-Based Gradient Descent

1. By the Lipschitz continuity in $\infty$-norm, we have

$$\begin{aligned}
f(x^{t+1}) &\leq f(x^t) + \nabla f(x^t)^T (x^{t+1} - x^t) + \frac{L_\infty}{2} \|x^{t+1} - x^t\|_\infty^2 \\
&= f(x^t) + \nabla f(x^t)^T \Big( - \frac{\|\nabla f(x^t)\|_1}{L_\infty} sign(\nabla f(x^t)) \Big) + \frac{L_\infty}{2} \Big\| - \frac{\|\nabla f(x^t)\|_1}{L_\infty} sign(\nabla f(x^t)) \Big\|_\infty^2 \\
&= f(x^t) - \frac{1}{L_\infty} \|\nabla f(x^t)\|_1^2 + \frac{1}{2L_\infty} \|\nabla f(x^t)\|_1^2 \\
&= f(x^t) - \frac{1}{2L_\infty} ||\nabla f(x^t)||_1^2 \\
&\leq f(x^t) - \frac{1}{2L_\infty} ||\nabla f(x^t)||_2^2
\end{aligned}$$

By the strong convexity, we have

$$-||\nabla f(x^t)||_2^2 \leq -2\mu(f(x^t) - f(x^*)).$$

Therefore, we get $f(x^{t+1}) - f(x^*) \leq (1 - \frac{u}{L_\infty})(f(x^t) - f(x^*))$.

2. If $f$ is $L_\infty$ Lipschitz continuous,

$$\begin{aligned}
\|f'(x) - f'(y)\|_2 &\leq \|f'(x) - f'(y)\|_1 \\
&\leq L_\infty \|x - y\|_\infty \\
&\leq L_\infty \|x - y\|_2
\end{aligned}$$

As $L$ is the minimum value of the Lipschitz $L_2$-norm constant, we have $L \leq L_\infty$. Similarly, if $f$ is $L_2$ Lipschitz continuous, we have

$$\begin{aligned}
\|f'(x) - f'(y)\|_1 &\leq \sqrt{d} \|f'(x) - f'(y)\|_2 \\
&\leq \sqrt{d} L \|x - y\|_2 \\
&\leq \sqrt{d} L \sqrt{d} \|x - y\|_\infty \\
&\leq dL \|x - y\|_\infty
\end{aligned}$$

As $L_\infty$ is the minimum value of the Lipschitz $L_\infty$-norm constant, we have $L_\infty \leq dL$. In conclusion, $L \leq L_\infty \leq dL$.

## 1.3 Block Coordinate Descent

1. By block-wise strong-smoothness and $p(b_t = b) = \frac{1}{k}$, we have

$$\begin{aligned}
f(x^{t+1}) &\leq f(x^t) - \frac{1}{L}\nabla_{b_t} f(x^t)^T(\nabla f(x^t) \circ e_{bt}) + \frac{1}{2L}||\nabla f(x^t) \circ e_{b_t}||^2 \\
&= f(x^t) - \frac{1}{2L}||\nabla f(x^t) \circ e_{b_t}||^2 \\
\mathbf{E}[f(x^{t+1})] &\leq \mathbf{E}[f(x^t) - \frac{1}{2L}||\nabla f(x^t) \circ e_{b_t}||^2] \\
&= \frac{1}{k}\sum_{b=1}^{k}(f(x^t) - \frac{1}{2L}||\nabla f(x^t) \circ e_{b_t}||^2) \\
&= f(x^t) - \frac{1}{2kL}\sum_{b=1}^{k}||\nabla f(x^t) \circ e_{b_t}||^2 \\
&= f(x^t) - \frac{1}{2kL}||\nabla f(x^t)||^2
\end{aligned}$$

From strong convexity, we can re-write the inequality into:

$$\begin{aligned}
\mathbf{E}(f(x^{t+1})) - f(x^*) &\leq f(x^t) - f(x^*) - \frac{u}{kL}[f(x^t) - f(x^*)] \\
&= (1 - \frac{u}{kL})[f(x^t) - f(x^*)]
\end{aligned}$$

2. Assuming that each block $b$ has its own strong-smoothness constant $L_b$ and the blocks are sampled proportional to $L_{b_t}$, we have

$$\begin{aligned}
f(x^{t+1}) &\leq f(x^t) - \frac{1}{L_{b_t}}\nabla_{b_t} f(x^t)^T(\nabla f(x^t) \circ e_{bt}) + \frac{1}{2L_{b_t}}||\nabla f(x^t) \circ e_{b_t}||^2 \\
&= f(x^t) - \frac{1}{2L_{b_t}}||\nabla f(x^t) \circ e_{b_t}||^2 \\
\mathbf{E}[f(x^{t+1})] &\leq \mathbf{E}[f(x^t) - \frac{1}{2L_{b_t}}||\nabla f(x^t) \circ e_{b_t}||^2] \\
&= \sum_{b=1}^{k}\frac{L_{b_t}}{\sum L_{b_t}}(f(x^t) - \frac{1}{2L_{b_t}}||\nabla f(x^t) \circ e_{b_t}||^2) \\
&= f(x^t) - \sum_{b=1}^{k}\frac{1}{2\sum L_{b_t}}||\nabla f(x^t) \circ e_{b_t}||^2 \\
&\leq f(x^t) - \frac{u}{\sum L_{b_t}}[f(x^t) - f(x^*)] \\
&= (1 - \frac{u}{\sum L_{b_t}})[f(x^t) - f(x^*)]
\end{aligned}$$

The second last line is from applying the strong convexity block-wise. Since $L = \max_{b_t}\{L_{b_t}\}$, $L_{b_t} \leq L, \sum L_{b_t} \leq kL$. If there exists some $L_{b_t} \neq L$, $1 - \frac{u}{\sum L_{b_t}} \leq 1 - \frac{u}{kL}$, meaning we have a faster convergence rate.

# 2 Large-Scale Algorithms

## 2.1 Coordinate Optimization

1. The final function value is 1.4724e+02 and the total time is 2.328713 seconds.

```
function [model] = logisticL2(X,y,lambda)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];
d = d+1;

% Initial values of regression parameters
w = zeros(d,1);

% Optimizaion parameters
maxPasses = 500;
progTol = 1e-4;
L = .25*max(sum(X.^2, 1))+ lambda;

w_old = w;

Xw = X*w;

for t = 1:maxPasses*d

    % Choose variable to update 'j'
    j = randi(d);

    % Compute partial derivative 'g_j'

    yXw = y.*(Xw);
    sigmoid = 1./(1+exp(-yXw));
    g_j = -(y.*(1-sigmoid))'*X(:,j) + lambda*w(j);

    % Update variable
    w(j) = w(j) - (1/L)*g_j;
    Xw = Xw - (1/L)*X(:,j)*g_j;

    % Check for lack of progress after each "pass"
    if mod(t,d) == 0
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',t/d,
            logisticL2_loss(w,X,y,lambda),change);
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
```

```
        end
        w_old = w;
    end
end


model.w = w;
model.predict = @predict;
end
```

2. The final function value is 1.4418e+02 and the total number of passes is 153.

```
function [model] = logisticL2(X,y,lambda)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];
d = d+1;

% Initial values of regression parameters
w = zeros(d,1);

% Optimizaion parameters
maxPasses = 500;
progTol = 1e-4;
L_candidate = .25*(sum(X.^2, 1))+ lambda;
w_old = w;

Xw = X*w;

for t = 1:maxPasses*d

    % Choose variable to update 'j'
    %j = randi(d);

    p = L_candidate/(sum(L_candidate));
    j = sampleDiscrete(p);
    L = L_candidate(j);

    % Compute partial derivative 'g_j'

    yXw = y.*(Xw);
    sigmoid = 1./(1+exp(-yXw));
    g_j = -(y.*(1-sigmoid))'*X(:,j) + lambda*w(j);

    %g_j = g(j);

    % Update variable
    w(j) = w(j) - (1/L)*g_j;
    Xw = Xw - (1/L)*X(:,j)*g_j;

    % Check for lack of progress after each "pass"
    if mod(t,d) == 0
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',t/d,
```

```
            logisticL2_loss(w,X,y,lambda),change);
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
        end
        w_old = w;
    end
end


model.w = w;
model.predict = @predict;
end
```

3. The final function value is 1.4091e+02 and the total number of passes is 149.

```
function [model] = logisticL2(X,y,lambda)

% Add bias variable
[n,d] = size(X);
X = [ones(n,1) X];
d = d+1;

% Initial values of regression parameters
w = zeros(d,1);

% Optimizaion parameters
maxPasses = 500;
progTol = 1e-4;
L_candidate = .25*(sum(X.^2, 1))+ lambda;
w_old = w;

Xw = X*w;

for t = 1:maxPasses*d

    % Choose variable to update 'j'
    j = randi(d);
    L = L_candidate(j);

    % Compute partial derivative 'g_j'

    yXw = y.*(Xw);
    sigmoid = 1./(1+exp(-yXw));
    g_j = -(y.*(1-sigmoid))'*X(:,j) + lambda*w(j);

    % Update variable
    w(j) = w(j) - (1/L)*g_j;
    Xw = Xw - (1/L)*X(:,j)*g_j;

    % Check for lack of progress after each "pass"
    if mod(t,d) == 0
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',t/d,
            logisticL2_loss(w,X,y,lambda),change);
```

```
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
        end
        w_old = w;
    end
end


model.w = w;
model.predict = @predict;
end
```

4. We stated in class that $L_{j_t}$ is used as the step size and sampling $j_t$ proportional to $L_{j_t}$ gives

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

By using the uniform sampling, $E[\bar{L}]_{Uniform} = \frac{1}{d}\sum_{i=1}^{d} L_i$

By using the Lipschitz sampling, $E[\bar{L}]_{Lipschitz} = \sum_{i=1}^{d} p_i L_i = \frac{1}{\sum_i^d L_i}\sum_{i=1}^{d} L_i^2$

To prove that uniform sampling has a tighter bound than Lipschitz sampling, we need to show that

$$E[\bar{L}]_{Uniform} \leq E[\bar{L}]_{Lipschitz},$$

which is equivalent to showing

$$\frac{1}{d}\left(\sum_{i=1}^{d} L_i\right)^2 \leq \sum_{i=1}^{d} L_i^2.$$

According to the CauchySchwarz inequality, the above inequality holds and thus uniform sampling outperforms Lipschitz sampling.

## 2.2 Proximal-Gradient

1. errTrain = 0.0060; errTest = 0.2740
The number of non-zero parameters: 500
The number of original features that the model uses: 100

```
function [model] = softmaxClassifierL2(X,y,lambda)

% 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,lambda);

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,lambda)

[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))+ sum(sum((lambda/2)*W.^2)) ;


g = zeros(p,k);
for c = 1:k
    g(:,c) = X'*(exp(XW(:,c))./Z-(y == c)) + lambda*W(:,c);
end

g = reshape(g,[p*k 1]);
end
```

2. The number of non-zero parameters: 35
   The number of original features that the model uses: 19

```
function [model] =  softmaxClassifierL1(X,y,lambda,maxIter)

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

W = zeros(d,k); % Each column is a classifier
W(:) = proxGradL1(@softmaxLoss,W(:),X,y,k,lambda,maxIter);


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

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


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

[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
```

3. errTrain = 0.0220; errTest = 0.0540 The number of non-zero parameters: 115
   The number of original features that the model uses: 23

```
function [model] = softmaxClassiferGL1(X,y,lambda,maxIter)

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

W = zeros(d,k); % Each column is a classifier
W(:) = proxGradGroupL1(@softmaxLoss,W(:),X,y,k,lambda,maxIter);


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

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


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

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

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

ind = sub2ind([n k],[1:n]',y);

tmp = sqrt(sum(W.^2,2));

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

function [w,f] = proxGradGroupL1(funObj,w,X,y,k,lambda,maxIter)
% Minimize funOb(w) + lambda*sum(abs(w))

% Evaluate initial objective and gradient of smooth part
[f,g] = funObj(w,X,y,k,lambda);
funEvals = 1;

L = 1;
[n,p] = size(X);
k = max(y);
while funEvals < maxIter

    % proximal-gradient step
    alpha = 1/L;
    W =  reshape(w,[p k]);
    W_new = W;
    W_update = reshape(w - alpha*g,[p k]);
    for i = 1: p
        W_new(i,:) =  softThreshold(W_update(i,:),alpha*lambda);
    end
    w_new = W_new(:);
    %w_new = softThreshold(w - alpha*g,alpha*lambda);
    [f_new,g_new] = funObj(w_new,X,y,k,lambda);
    funEvals = funEvals + 1;

    % adaptive step-size
    while f_new > f + g'*(w_new - w) + (L/2)*norm(w_new-w)^2
        L = L*2;
        alpha = 1/L;
        w_new = softThreshold(w - alpha*g,alpha*lambda);
        [f_new,g_new] = funObj(w_new,X,y,k,lambda);
        funEvals = funEvals + 1;
    end

    w = w_new;
    f = f_new;
    g = g_new;

    % Print out how we are doing
    optCond = norm(w-softThreshold(w-g,lambda),'inf');
    fprintf('%6d %15.5e %15.5e %15.5e\n',funEvals,alpha,f + lambda*sum(abs(w)
        ),optCond);

    if optCond < 1e-1
        break;
    end
end
end

function [w] = softThreshold(w,threshold)
    w = (w./(norm(w))).*max(0,norm(w)-threshold);
end
```

```matlab
function [nll,g] = softmaxLoss(w,X,y,k,lambda)

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

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

ind = sub2ind([n k],[1:n]',y);

tmp = sqrt(sum(W.^2,2));

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
```

## 2.3 Stochastic Gradient

1. We tried the step-sizes defined by $1/(0.1t)$,$1/(0.2t)$..., $1/(2t)$, and $1/(0.1\sqrt{t})$,$1/(0.2\sqrt{t})$..., $1/(2\sqrt{t})$. The maximum number of passes is set to 10. The step-size with the best performance is $\frac{1}{1.8\sqrt{t}}$ and its corresponding function value is 2.7145e+04.

2. We choose to weight the $w$ with equal weights. The step-size with the best performance is $1/\sqrt{t}$ and its corresponding function value is 2.7076e+04.

3. We tried the step-sizes defined by $1/(0.1\sqrt{t})$,$1/(0.2\sqrt{t})$..., $1/(2\sqrt{t})$, and $\delta$ defined by $1, 101, 201, 301, 401$. The sequence with the best performance is step-size $= 1/(0.1\sqrt{t})$ and $\delta = 301$, the function value is 2.7083e+04. There are other combinations that gives close results: e.g. step-size $= 1/(0.2\sqrt{t})$ and $\delta = 301$; step-size $= 1/(0.3\sqrt{t})$ and $\delta = 301$.

```matlab
load quantum.mat
[n,d] = size(X);
lambdaFull = 1;

% Initialize
maxPasses = 10;
progTol = 1e-4;
w = zeros(d,1);
lambda = lambdaFull/n; % The regularization parameter on one example
dt = zeros(d,d);
tmp =  zeros(d,1); % save the summation of the squared gradient in D
% Stochastic gradient
w_old = w;
c_candidate = 0.1:0.1:2;
for j = 1: length(c_candidate)   %search for the constant value in the step
    sie
    for k = 1:100:500    %search for the optimal delta
```

```
        w = zeros(d,1);
        w_old = w;
        w_pre_iter = w;
        tmp =  zeros(d,1);
        delta = k;
        c_candidate(j);
            for t = 1:maxPasses*n

                % Choose variable to update
                i = randi(n);

                % Evaluate the gradient for example i

                [f,g] = logisticL2_loss(w,X(i,:),y(i),lambda);
                tmp = tmp + g.^2;
                D = diag(1./(sqrt(delta + tmp)));

                % Choose the step-size
                alpha = 1./(c_candidate(j)*t^(0.5));

                w = w - alpha*D*g;

                if mod(t,n) == 0
                    change = norm(w-w_old,inf);
                    %fprintf('Passes = %d, function = %.4e, change = %.4f\n', t/n,logisticL2_loss(w,X,y,lambdaFull),change);
                    if change < progTol
                    %    fprintf('Parameters changed by less than progTol on pass\n');
                        break;
                    end
                    w_old = w;
                end
            end
             fprintf('c = %d,delta = %d, Passes = %d, function = %.4e \n', c_candidate(j),k,t/n,logisticL2_loss(w,X,y,lambdaFull));
    end

end
```

4. The function value is 2.7068e+04 after 10 passes. It appears to converge faster than the algorithms in 1-3.

```
load quantum.mat
[n,d] = size(X);
lambdaFull = 1;

% Initialize
maxPasses = 10;
progTol = 1e-6;
w = zeros(d,1);
lambda = lambdaFull/n; % The regularization parameter on one example

% Stochastic gradient
w_old = w;
```

```
L = .25*max(sum(X.^2,2)) + lambda;

D = zeros(d,1);   %represent the sum of the gradients calculated from n samples
Y_i = zeros(n,d);
g_old = zeros(d,1);
for t = 1:maxPasses*n

    % Choose variable to update
    i = randi(n);

    % Evaluate the gradient for example i

    [f,g] = logisticL2_loss(w,X(i,:),y(i),lambda);

    D = D - (Y_i(i,:))' + g;
    Y_i(i,:) = g';

    % Choose the step-size
    alpha = 1/L;
    w = w - (alpha*D./n);

    if mod(t,n) == 0
        change = norm(w-w_old,inf);
        fprintf('Passes = %d, function = %.4e, change = %.4f\n',t/n,logisticL2_loss(w,X,y,lambdaFull),change);
        if change < progTol
            fprintf('Parameters changed by less than progTol on pass\n');
            break;
        end
        w_old = w;
    end
end
```

# 3 Kernels and Duality

## 3.1 Fenchel Duality

1. Let $X = X, z = X\omega$.

$$\begin{aligned}
f^*(u) &= \sup_z \Big\{ u^T z - \frac{1}{2}\|z-y\|^2 \Big\} \\
&= \sup_z \Big\{ -\frac{1}{2}\|z-u-y\|^2 + \frac{1}{2}\|u\|^2 + u^T y \Big\} \\
&= \frac{1}{2}\|u\|^2 + u^T y. \\
g^*(u) &= \sup_\omega \Big\{ u^T \omega - \frac{\lambda}{2}\|\omega\|^2 \Big\} \\
&= \sup_\omega \Big\{ -\frac{\lambda}{2}\|\omega - \frac{1}{\lambda}u\|^2 + \frac{1}{2\lambda}\|u\|^2 \Big\} \\
&= \frac{1}{2\lambda}\|u\|^2.
\end{aligned}$$

Therefore, we have

$$D(z) = -f^*(-z) - g^*(X^T z) = -(\frac{1}{2}\|z\|^2 - z^T y) - \frac{1}{2\lambda}\|X^T z\|^2.$$

2. Let $X = X, z = X\omega$.

$$\begin{aligned}
f^*(u) &= \sup_z \Big\{ u^T z - \|z-y\|_1 \Big\} \\
&= \sup_z \Big\{ \sum_{i=1}^n \big( u_i z_i - |z_i - y_i| \big) \Big\} \\
&= \begin{cases} u^T y & \text{if } |u_i| \le 1 \text{ for all } i \\ \infty & \text{else} \end{cases} \\
g^*(u) &= \sup_\omega \Big\{ u^T \omega - \lambda\|\omega\|_1 \Big\} \\
&= \begin{cases} 0 & \text{if } |u_i| \le \lambda \text{ for all } i \\ \infty & \text{else} \end{cases}
\end{aligned}$$

Therefore, we have

$$D(z) = -f^*(-z) - g^*(X^T z) = \begin{cases} z^T y & \text{if } |z_i| \le 1 \text{ and } |(X^T z)_i| \le \lambda \text{ for all } i \\ -\infty & \text{else} \end{cases}.$$

3. Let $X = -\begin{pmatrix} y^1x^1 \\ y^2x^2 \\ \vdots \\ y^nx^n \end{pmatrix}$ and $z = X\omega$.

$$
\begin{aligned}
f^*(u) &= \sup_z \Big\{ u^T z - \sum_{i=1}^n \log(1+\exp(z_i)) \Big\} \\
&= \begin{cases} \sum_{i=1}^n \Big\{ u_i \log(u_i) + (1-u_i)\log(1-u_i) \Big\} 1_{(0<u_i<1)} & \text{if } 0 \le u_i < 1 \text{ for all } i \\ \infty & \text{else} \end{cases} \\
g^*(u) &= \sup_\omega \Big\{ u^T\omega - \frac{\lambda}{2}\|\omega\|^2 \Big\} \\
&= \sup_\omega \Big\{ -\frac{\lambda}{2}\|\omega - \frac{1}{\lambda}u\|^2 + \frac{1}{2\lambda}\|u\|^2 \Big\} \\
&= \frac{1}{2\lambda}\|u\|^2.
\end{aligned}
$$

Therefore, we have

$$
\begin{aligned}
&D(z) = -f^*(-z) - g^*(X^T z), \\
&\quad = \begin{cases} -\sum_{i=1}^n \Big\{ -z_i \log(-z_i) + (1+z_i)\log(1+z_i) \Big\} 1_{(-1<z_i<0)} - \frac{1}{2\lambda}\|X^T z\|^2, & \text{if } -1 < z_i \le 0 \text{ for all } i \\ -\infty & \text{else} \end{cases}
\end{aligned}
$$

.

## 3.2 Stochastic Dual Coordinate Ascent

The code is given below. The optimal values of the primal and dual objectives with $\lambda = 1$ are 90.6829 and 90.6827 respectively. The number of support vectors is 99.

```
%%%%%%%%%%%%%%%%%%% Matlab code
X = load('statlog.heart.data');
y = X(:,end);
y(y==2) = -1;
X = X(:,1:end-1);
n = size(X,1);

% Add bias and standardize
X = [ones(n,1) standardizeCols(X)];
d = size(X,2);

% Set regularization parameter
lambda = 1;

% Some values used by the dual
```

```
YX = diag(y)*X;
G = YX*YX';

% Find min for -D(z)
z = quadprog(G/lambda, -ones(n,1), [],[],[],[], zeros(n,1), ones(n,1));

% Convert from dual to primal variables
w = (1/lambda)*(YX'*z);

% Evaluate primal objective:
P = sum(max(1-y.*(X*w),0)) + (lambda/2)*(w'*w)

% Evaluate dual objective:
D = sum(z) - (z'*G*z)/(2*lambda)

% count number of support vectors
sum(z> 0.01)
%%%%%%%%%%%%%%%%%%% end-Matlab code
```

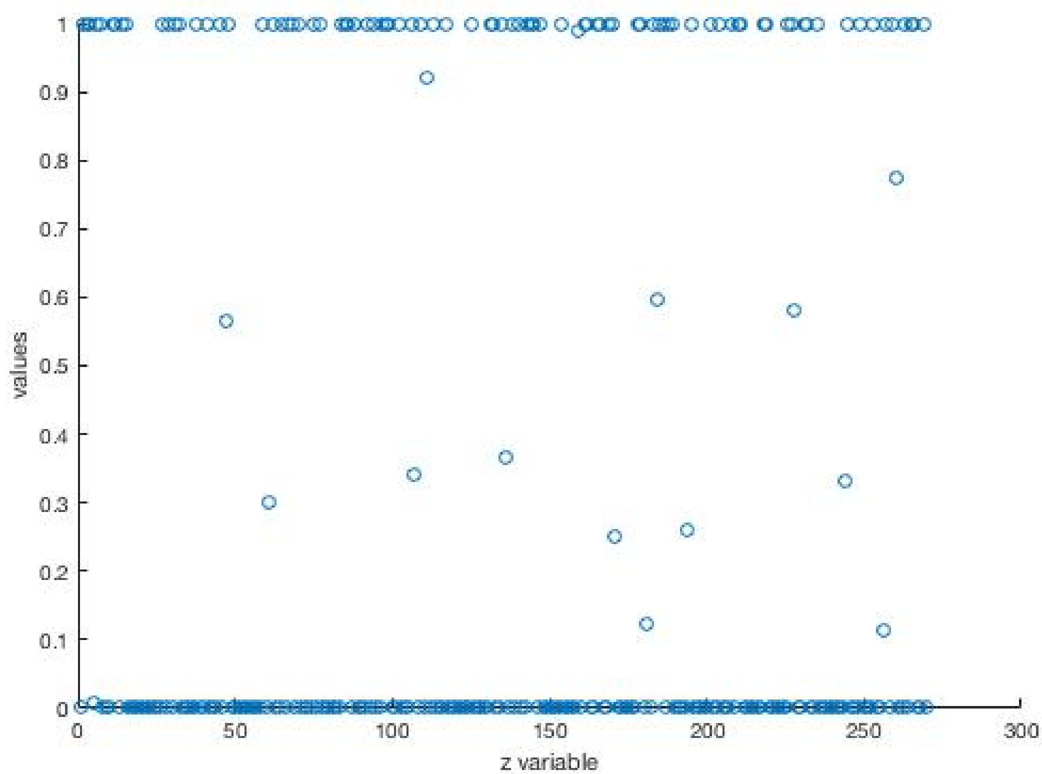


Figure 1: Values of $z$ variables.

### 3.3 Large-Scale Kernel Methods

1.The best value for $\sigma$ is 0.5 and the best value for $\lambda$ is $\lambda_1 = 4.8828e-04$. The $\sigma$ with Gaussian kernels is the same as the one with Gaussian RBFs. The $\lambda$ for Gaussian RBFs was $\lambda_2 = 1.47e-05$. The approximate relationship between $\lambda_1$ and $\lambda_2$ is that $\lambda_2 \approx \lambda_1 * mean(K(:))$, where $K = rbfBasis(X, X, sigma)$ is the matrix of pair-wise evaluations of the kernel for all training patterns.

2. The code is provided below and the performances with different $m$ values ($m = 10, 30, 40, 60, 200$) are visually examined by Figures 2-6. From the figures we see that the performance gets better when the $m$ value increases. The performance of m=60 is as good as using the whole dataset (i.e. m=200). The squaredTestError values are 2.0787e+04 with m=10, 3.6422e+03 with m=30, 429.0070 with m=40, 82.8489 with m=60, and 83.4175 with m=200.

```
%%%%%%%%%%%%%%%%% Matlab code
lambda=4.8828e-04;
sigma = 0.5;
m=10;  % or 30, 40, 60, 200
model = kernelRegression332(X,y,lambda,sigma,m);
yhat = model.predict(model, Xtest,m);
squaredTestError = sum((yhat-ytest).^2)/t

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, m);   % Gaussian kernel
plot(Xhat,yhat,'r');
ylim([-300 400]);


function [model] = kernelRegression332(X,y,lambda,sigma, m)

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

  % new
  K11 = rbfBasis(X(1:m,:),X(1:m,:),sigma);
  K21 = rbfBasis(X((m+1):n,:),X(1:m,:),sigma);
  K1 = [K11;K21];
  z=(K1'*K1+lambda*K11)\K1'*y;

  model.X = X;
  model.z = z;
  model.sigma = sigma;
```

```
  model.predict = @predict;
end

function [yhat] = predict(model,Xhat,m)
  Khat = rbfBasis(Xhat,model.X(1:m,:),model.sigma);
  yhat = Khat*model.z;
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
%%%%%%%%%%%%%%%%%%% end- Matlab code
```

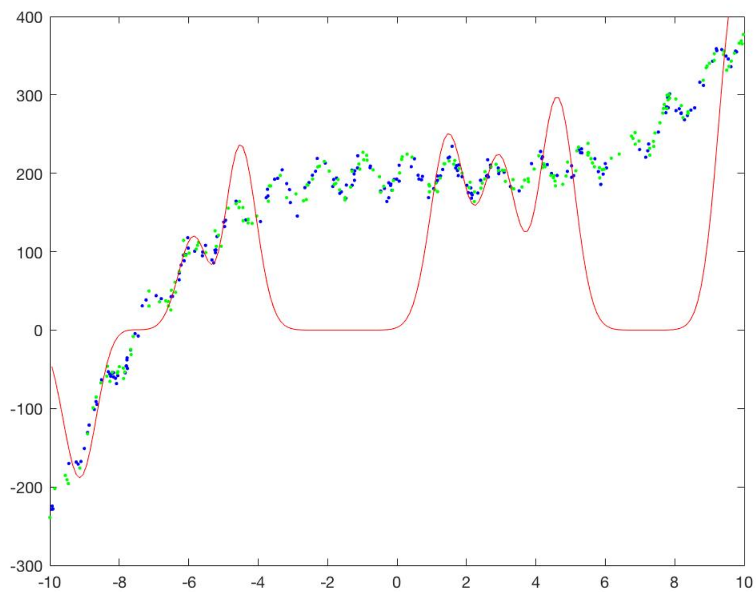


Figure 2: Performance with $m = 10$.

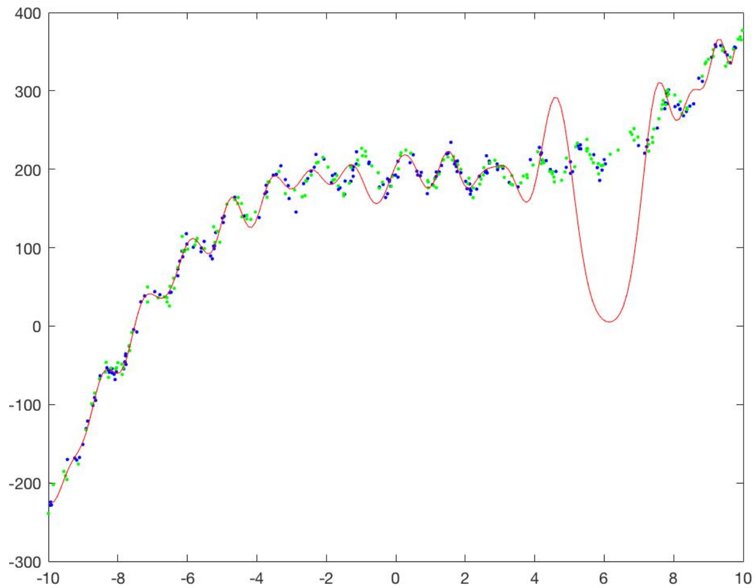


Figure 3: Performance with $m = 30$.

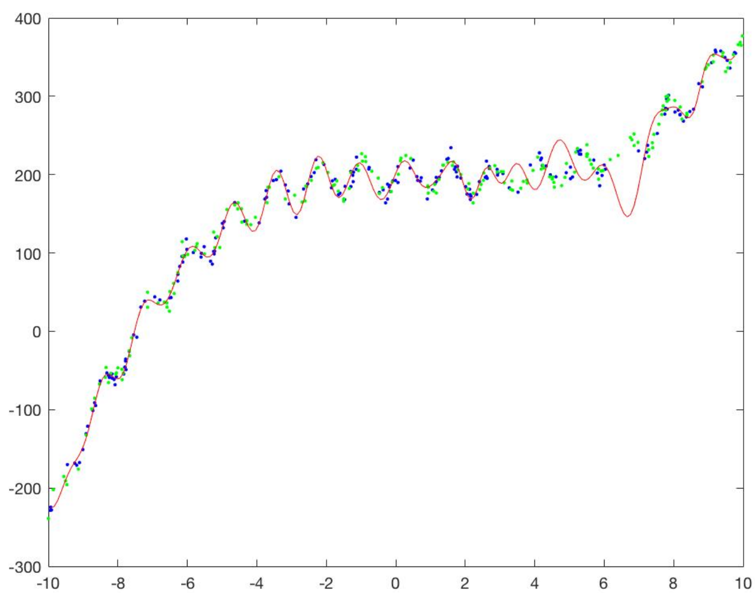


Figure 4: Performance with $m = 40$.

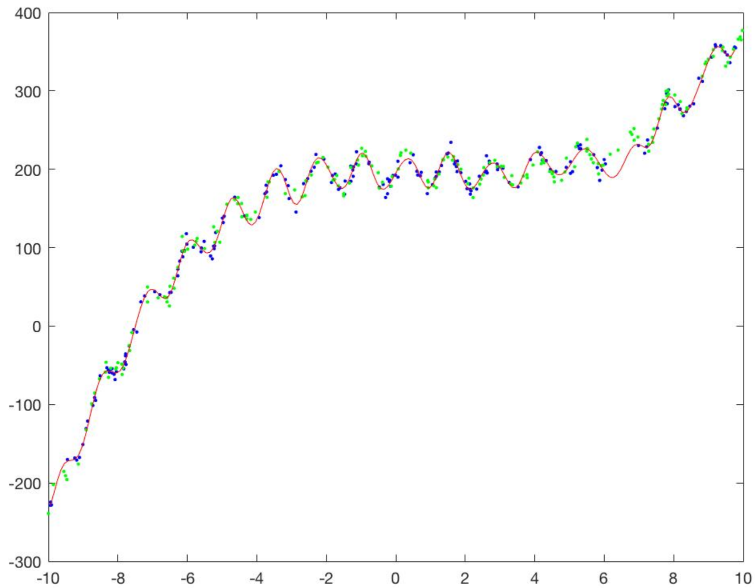


Figure 5: Performance with $m = 60$.

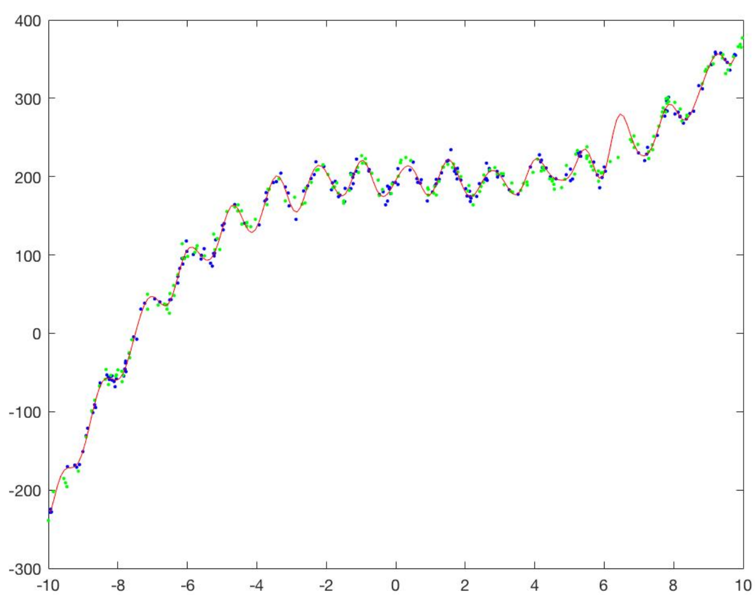


Figure 6: Performance with $m = 200$.

3. The code is given below and the performances with different $m$ values ($m = 3, 5, 8, 10, 20$) are visually examined by Figures 7-11. From the figures we see that the performance gets better when the $m$ value increases. This method can achieve a fitting curve that captures the overall pattern well with a small value of $m$ (m=10), which is much smaller than the $m$ value of the

subset of regressors model (m=60). However, this method does not capture the wiggly pattern in the data as the subset of regressors method does.

```
%%%%%%%%%%%%%%%%%%% Matlab code
clear all
close all

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

lambda=4.8828e-04;
sigma = 0.5;

rng(1);
m=3;  % or 5, 8, 10, 20
model = kernelRegression333(X,y,lambda,sigma,m);
yhat = model.predict(model, Xtest);
squaredTestError = sum((yhat-ytest).^2)/t
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]);


function [model] = kernelRegression333(X,y,lambda,sigma, m)

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

  R = randn(d, m)*sigma;  % dxm
  Z = exp(sqrt(-1)*X*R); % n,m

  % Solve least squares problem(
  z = (Z'*Z + lambda*eye(m))\Z'*y;  % m,1

  model.R = R;  % dxm
  model.z = z;  % mx1
  model.sigma = sigma;
  model.predict = @predict;
end
```

```
function [yhat] = predict(model,Xhat)
  Zhat = exp(sqrt(-1)*Xhat*model.R); % n2xm
  yhat = Zhat*model.z;
end
%%%%%%%%%%%%%%%%%%% end- Matlab code
```

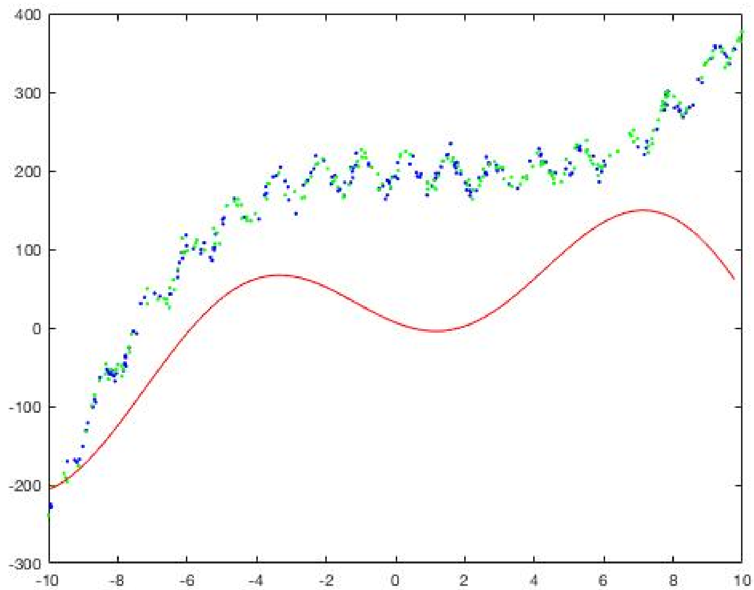


Figure 7: Performance with $m = 3$.

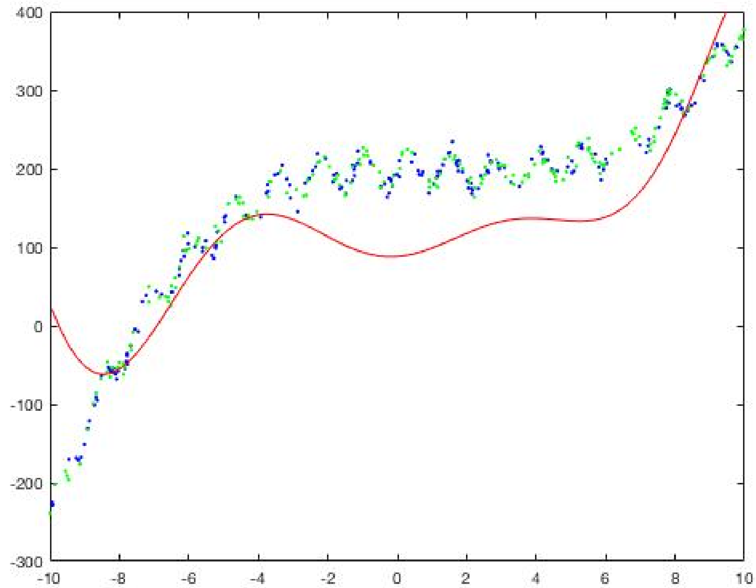


Figure 8: Performance with $m = 5$.

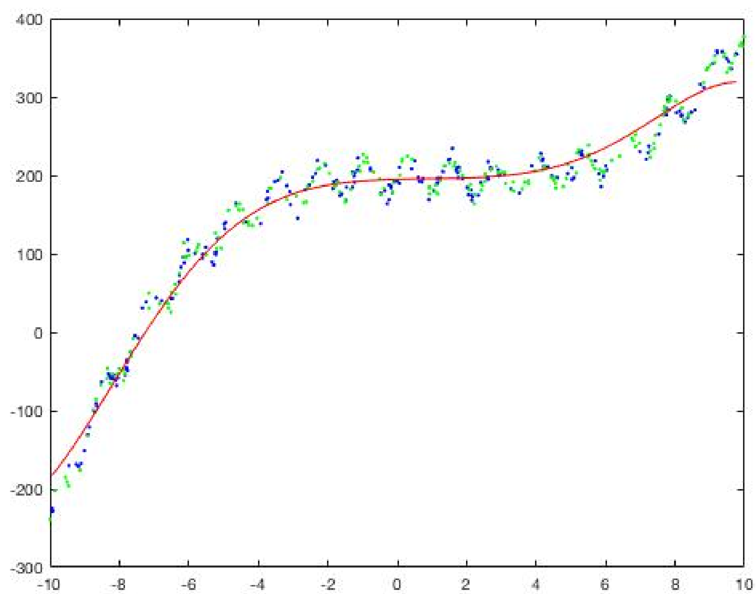


Figure 9: Performance with $m = 8$.

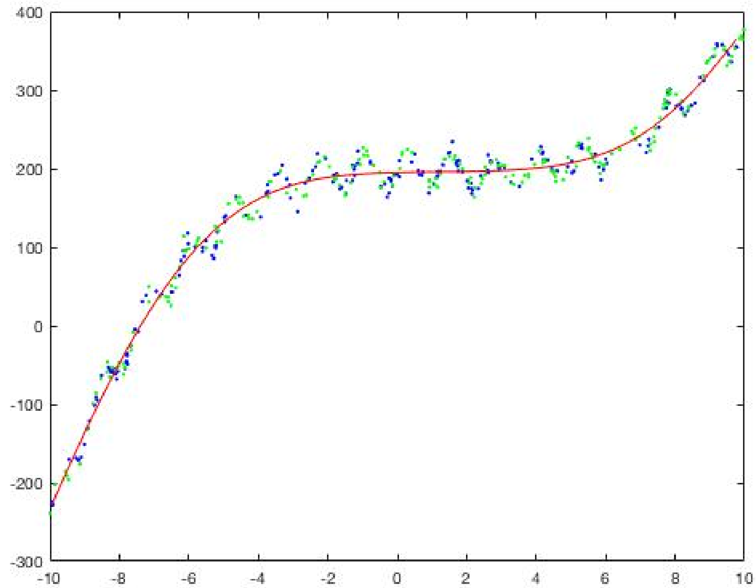


Figure 10: Performance with $m = 10$.

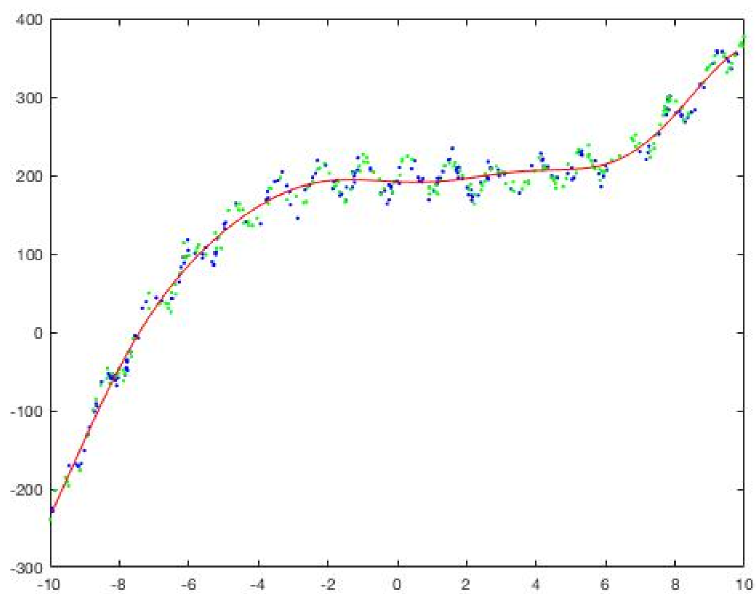


Figure 11: Performance with $m = 20$.