CSE512 HW5

Tim Zhang (110746199)

1 Report

Submission team: modulo
Kaggle user name: modulo
—
My first attempts at learning involved SGD using SVM and squared hinge loss. This turned out to perform very poorly and I was unable to breach even 55%. I then tried mapping the features to a polynomial space. I accomplished this by raising each individual feature by k and also computing the cross terms of corresponding features between the two individuals in a sample. I then raised the cross terms to the power k.

Using this feature mapping scheme and oversampling of the unbalanced data set I was able to pass the professors score using Least Squares and $k = 3$. I further improved on my score using regularized least squares with $k = 5$.

Some other approaches I tried before polynomial least squares were logistic regression and regular hinge loss SVM. I also implemented undersampling and mini-batch sizes for SGD.

2 Implementation

NOTE: I had to introduce newlines for some code which ran off of the page for LaTeX formatting. These are not present in the actual code.

```
%---------------------------------------------
% Main()
%---------------------------------------------
function driver(loss, training_path, test_path, k, C, a, T, B, validation_size, keep,
sample, regularizer, lambda, Xtrain, Xtest, ytrain)
    [Xtrain, Xtest, ytrain] = formatData(training_path, test_path, k, sample);

    % Run algorithm
    if strcmp('Least Squares', loss)
        [w] = train_ls(Xtrain, ytrain, regularizer, lambda);
        ypredicted = predict_ls(Xtest, w);
    else
        [w, validation_error] = train_sgd(loss, Xtrain, ytrain, C, a, T, B,
        validation_size, keep);
        ypredicted = predict_sgd(Xtest, w, loss);
    end

    [mte, ~] = size(Xtest);

    % Format output for submission
    fileID = fopen('prediction.txt', 'w');
    fprintf(fileID, 'Id,Prediction\n');

    M = (1:mte);

    for i = 1:mte
        fprintf(fileID, '%g,%g\n', M(i), ypredicted(i));
    end

    disp('Results saved');
end

%
% Formats data
%
function [Xtrain, Xtest, ytrain] = formatData(training_path, test_path, k, sample)
```

```
    % Put data files into appropriate matrices
    disp('Parsing data files...');
    [Xtrain, ytrain] = parse(training_path);
    [Xtest, ~] = parse(test_path);

    % Undersample negative class
    disp('Balancing dataset...');
    if strcmp('Oversample', sample)
        [Xtrain, ytrain] = oversample(Xtrain, ytrain);
    else
        [Xtrain, ytrain] = undersample(Xtrain, ytrain);
    end

    % Apply polynomial basis function to the degree k
    disp('Generating polynomial features...');
    Xtrain = generate_poly_features(Xtrain, k);
    Xtest = generate_poly_features(Xtest, k);

    disp('Generating cross features...');
    Xtrain = generate_cross_features(Xtrain, k);
    Xtest = generate_cross_features(Xtest, k);

    % Normalize data
    disp('Normalizing data...');
    Xtrain = zscore(Xtrain);
    Xtest = zscore(Xtest);
end

%---------------------------------------------
% Learning Algorithms
%---------------------------------------------
%
% Implements Stochastic Subgradient Descent
% Can run as mini-batch mode with argument B
%
function [w, error] = train_sgd(loss, X, y, C, a, T, B, validation_size, keep)
    [m, d] = size(X);               % Dimensions of data
    w_t = zeros(d, 1);              % Initialize w_0 = 0
    total = zeros(d, 1);            % Average over w
    m_train = m - validation_size;  % Training size with validation removed
    run = 0;                        % Logs number of iterations
```

```
% Display algorithm details
disp('Training via Stochastic Subgradient Descent...');
disp('Loss:');
disp(loss);
disp('Iteration:');

for t = 1: T
    disp(t);  % Logging
    order = randperm(m_train, m_train);  % Shuffle data
    i = 1;  % Initialize index into training set to 1

    % Run algorithm over all training samples
    while i <= m_train

        eta = a/sqrt(i);  % Weight decay
        run = run + 1;  % Increase run counter

        % Calculate new weight vector using subgradient
        for j = i: i + B
            if j <= m_train
                r = order(j);

                % Hinge Loss
                if strcmp('SVM Hinge', loss)
                    if y(r) * (X(r, :) * w_t) <= 1
                        w_t = (1 - eta) * w_t + (eta * C * (y(r) * X(r, :)'));
                    else
                        w_t = (1 - eta) * w_t;
                    end

                % Squared Hinge Loss
                elseif strcmp('SVM Squared Hinge', loss)
                    if y(r) * (X(r, :) * w_t) <= 1
                        w_t = w_t + (eta * C * ((2 * y(r) * X(r, :)')
                        - (2 * (X(r, :)' * (X(r, :) * w_t)))));
                    end

                % Logistic Regression
                elseif strcmp('Logistic Regression', loss)
                    % Convert label to {0, 1}
```

```
                    if y(j) == 1
                        y_l = 1;
                    else
                        y_l = 0;
                    end

                    p = 1/(1 + exp(-1 * (X(r, :) * w_t)));
                    w_t = w_t + ((eta * (y_l - p)) * X(r, :)');

                else
                    disp('Invalid algorithm');
                end
            end
        end

        i = i + B;

        % Only keep the last "keep" points
        if run > (T * ceil(m_train/B)) - keep
            total = total + w_t;
        end
    end
end

w = (1 / keep) .* total;  % Average over the kept weights


% Calculate training error
disp('Calculating training error...');
error = 0;
pred = zeros(m_train, 1);
j = 1;

for i = 1: m_train
    if strcmp('Logistic Regression', loss)
        p = 1/(1 + exp(-1 * (X(i, :) * w)));

        if (p > .5) && (y(i) == -1)
            error = error + 1;
        elseif (p <= .5) && (y(i) == 1)
            error = error + 1;
```

```
        end

        disp(p);
        disp(y(i));

    else
        if y(i) * (X(i, :) * w) <= 0
            error = error + 1;
        end
    end

    pred(j) = X(i, :) * w;
    j = j + 1;
end

error = error / m_train;
disp(error);

disp('Calculating training AUC...');
[~, ~, ~, AUC] = perfcurve(y, pred, 1);
disp(AUC)

% Calculate validation error
disp('Calculating validation error...');
error = 0;

for i = m_train: m
    if strcmp('Logistic Regression', loss)
        p = 1/(1 + exp(-1 * (X(i, :) * w)));

        if (p >= .5) && (y(i) == -1)
            error = error + 1;
        elseif (p < .5) && (y(i) == 1)
            error = error + 1;
        end

    else
        if y(i) * (X(i, :) * w) <= 0
            error = error + 1;
        end
    end
```

```
    end

    error = error / validation_size;
    disp(error);
end

%
% Computes predictions for SGD
%
function ypredicted = predict_sgd(Xte, w, loss)
    [m, ~] = size(Xte);
    ypredicted = m:1;
    positive = 0;

    disp('Predicting test data...');

    for i = 1: m
        if strcmp('Logistic Regression', loss)
            p = 1/(1 + exp(-1 * (Xte(i, :) * w)));

            if p >= .5
                positive = positive + 1;
                ypredicted(i) = 1;
            else
                ypredicted(i) = -1;
            end
        else
            if Xte(i, :) * w > 0
                positive = positive + 1;
            end

            ypredicted(i) = Xte(i, :) * w;
            %disp(ypredicted(i));
        end
    end

    disp('Positive matches');
    disp(positive);
end

%
```

```
% Least Squares
%
function [w] = train_ls(X, y, regularizer, lambda)
    [m, d] = size(X);

    if regularizer
        disp('Training Ridge Regression...');
        I = eye(d);
    else
        disp('Training Ordinary Least Squares...');
    end

    % Use regularized least squares
    if regularizer
        % Compute minimum weights
        w = inv((X.' * X) + (lambda * I)) * (X.' * y);

    % OLS
    else
        % Check for invertibility
        if det(X.' * X) == 0
            [V, D] = eig(X.' * X);
            D_plus = zeros(size(D));

            % Create the positive D matrix as discussed in lecture slides
            for i = 1:m
                for j = 1:d
                    if D(i, j) ~= 0
                        D_plus(i, j) = 1/D(i, j);
                    end
                end
            end

            w = V * D_plus * V.' * X.' * y;

        % Otherwise return the normal solution
        else
            w = X\y;
        end
    end
```

```
    disp(w);

    % Calculate training error
    disp('Calculating training error');
    error = 0;
    pred = zeros(m, 1);
    j = 1;

    for i = 1: m
        if y(i) * sign(X(i, :) * w) <= 0
            error = error + 1;
        end

        pred(j) = X(i, :) * w;
        j = j + 1;
    end

    error = error / m;
    disp(error);


    disp('Calculating training AUC...');
    [~, ~, ~, AUC] = perfcurve(y, pred, 1);
    disp(AUC)
end

%
% Computes predictions for Least Squares
%
function ypredicted = predict_ls(Xte, w)
    [m, ~] = size(Xte);
    ypredicted = m:1;

    disp('Predicting test data...');

    positive = 0;

    for i = 1: m
        pred = Xte(i, :) * w;

        if pred > 0
```

```
            positive = positive + 1;
        end

        ypredicted(i) = pred;
    end

    disp('Positive matches');
    disp(positive);
end

%--------------------------------------------
% Helper functions
%--------------------------------------------
%
% Parses data into X and Y
%
function [X, y] = parse(file)
    data = importdata(file, ',');

    y = data(:, end);
    data(:, end) = [];
    X = data;

    disp('Successfully parsed');
    disp(file);
end

%
% Function oversamples positive class to balance data
%
function [X, y] = oversample(Xtr, ytr)
    [m, d] = size(Xtr);
    pos = [];
    neg = [];

    % Split into positive and negative examples
    for i = 1: m
        if ytr(i) == 1
            pos = [pos i];
        else
            neg = [neg i];
```

```
        end
    end

    [~, m_pos] = size(pos);
    [~, m_neg] = size(neg);
    extra = ceil(m_neg / m_pos) - 1;

    X = zeros((m_pos * (extra + 1)) + m_neg, d);
    y = zeros((m_pos * (extra + 1)) + m_neg, 1);

    duplicates = zeros(1, (m_pos * (extra + 1)));  % Holds duplicate positives

    % Fill duplicate vector
    k = 1;
    for i = 1: m_pos
        for j = 1: extra + 1
            duplicates(k) = pos(i);
            k = k + 1;
        end
    end

    [~, m_dup] = size(duplicates);
    oversample = [duplicates neg];    % All indices into training data
    order = randperm(m_dup + m_neg);  % Random order of training data

    % Randomly fill X and y
    for i = 1: m_dup + m_neg
        r = oversample(order(i));  % Get the next index

        X(i, :) = Xtr(r, :);
        y(i) = ytr(r);
    end

end

%
% Function undersamples negative class to balance data
%
function [X, y] = undersample(Xtr, ytr)
    [m, d] = size(Xtr);
    pos = [];
```

```
    neg = [];

    % Split into positive and negative examples
    for i = 1: m
        if ytr(i) == 1
            pos = [pos i];
        else
            neg = [neg i];
        end
    end

    [~, m_pos] = size(pos);
    [~, m_neg] = size(neg);
    keep = randperm(m_neg);  % Randomly select negative examples to keep
    X = zeros(2 * m_pos, d);
    y = zeros(2 * m_pos, 1);
    undersample = zeros(1, m_pos);  % Holds the negative examples we keep

    % Fill undersample array
    for i = 1: m_pos
        undersample(i) = neg(keep(i));
    end

    new = [pos undersample];  % Holds the indices of all of the kept samples
    keep = randperm(2 * m_pos);

    % Fill X and y randomly
    for i = 1: 2 * m_pos
        X(i, :) = Xtr(new(keep(i)), :);
        y(i) = ytr(new(keep(i)));
    end
end

%
% Function generates poylnomial features to the degree k from input matrix
%
function X = generate_poly_features(X, k)
    [m, d] = size(X);
    new_features = zeros(m, d * k);
    X = [X new_features];
```

```
    for i = 1: m
        % Each feature is raised to the k
        poly_features = zeros(1, d * k);
        feature = 1;  % Counter to track position in poly_features

        % For each dimension
        for j = 1: d
            % Compute x^k terms
            for pow = 2: k
                poly_features(feature) = X(i, j)^pow;
                feature = feature + 1;
            end
        end

        X(i, d + 1:end) = poly_features;
    end
end

%
% Generates cross features
%
function X = generate_cross_features(X, k)
    [m, d] = size(X);
    individual_d = 58;
    new_features = zeros(m, individual_d * k);
    X = [X new_features];

    for i = 1: m
        poly_features = zeros(1, individual_d * k);
        term = 1;  % Index for poly_features

        % Compute cross-terms
        for j = 1: individual_d
            poly_features(term) = X(i, j) * X(i, j + individual_d);
            term = term + 1;
        end

        % Raise cross terms to power 2..k-1
        for t = 1: k - 1
            for j = 1: individual_d
                poly_features(term) = poly_features(term - (individual_d * t))^(t + 1);
```

```
                term = term + 1;
            end
        end

        X(i, d + 1:end) = poly_features;
    end
end
```