Christopher Boyd
216 869 356

EECS 4404: A4 - SGD and SoftSVM

Stochastic Gradient Descent

1a)

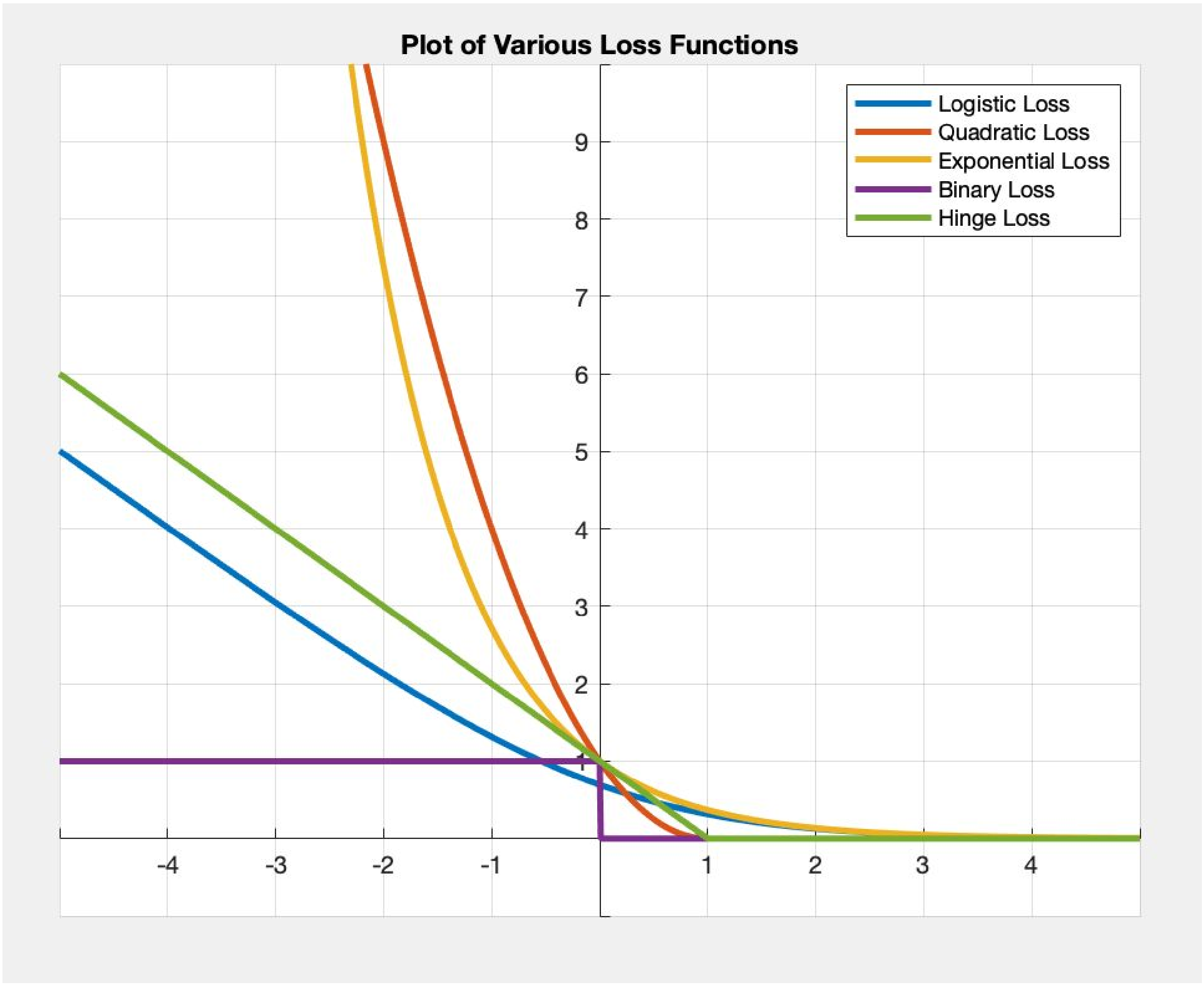


Fig 1. Comparison of Various Loss Functions

Of the presented loss functions, binary loss, stands apart. This is due to a prediction being either correct or incorrect. There is no distinction for the amount of correctness or incorrectness of a prediction. Compare this to all of the other loss functions (logistic, quadratic, exponential and hinge) where there is a sharp increase in loss as the level of incorrectness increases. Additionally, each of linear loss functions will show a small amount of loss even for correct predictions that fall close to the predictors hyperplane. This is seen in the small positive values for the graphs from 0 to ~1 for quadratic and hinge loss and from 0 ~ 2.5 for logistic and exponential losses. The linear loss functions differ in how quickly the loss increases as a prediction’s incorrectness increases (distance from the hyperplane). Quadratic and exponential loss functions increase at a much faster rate than the hinge and logistic loss functions.

1b)

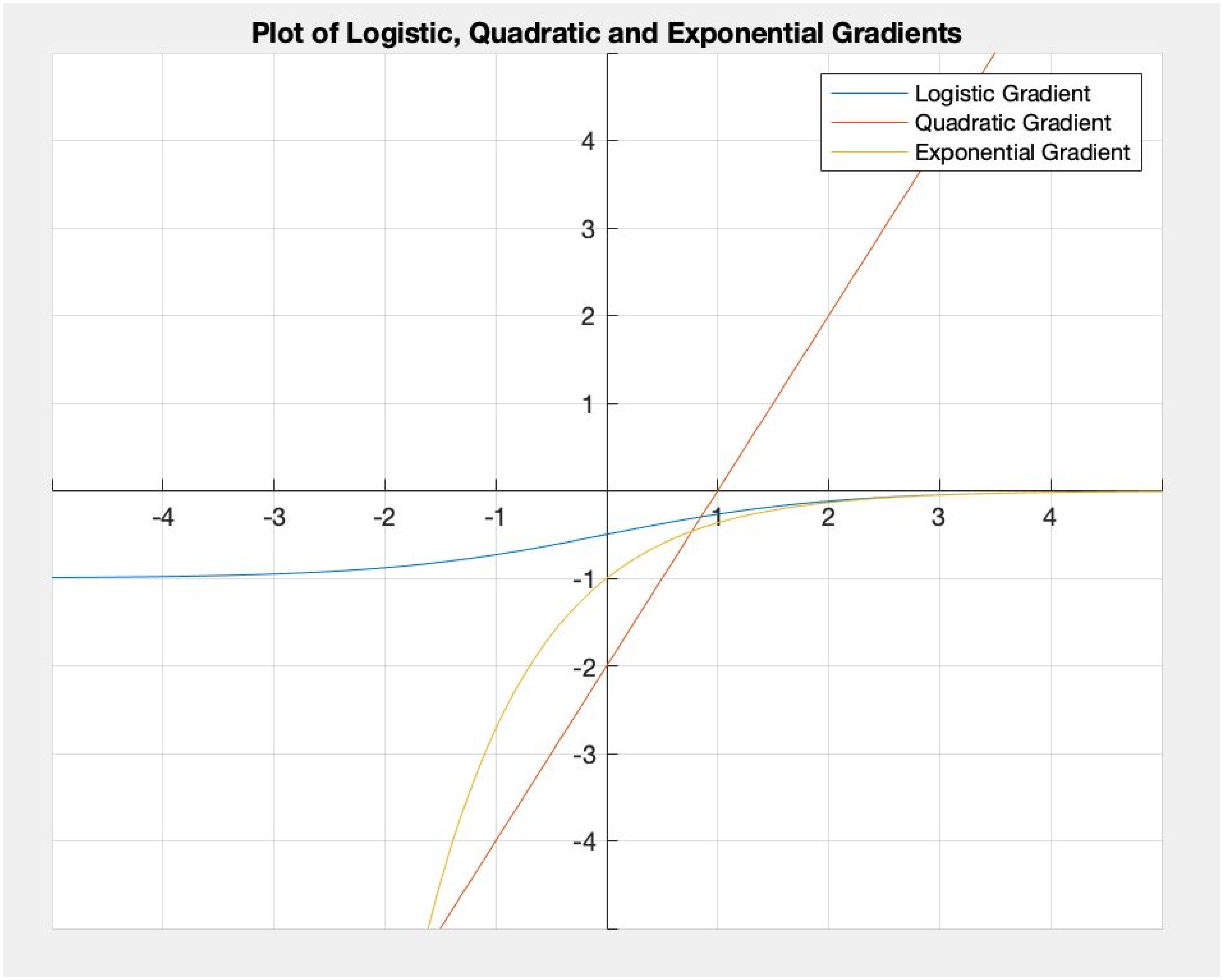


Fig 2. Loss Gradients for Logistic, Quadratic and Exponential Loss Functions

1c)

w_log = 2.5922, w_quad = 11.9027, w_exp = 74.4176

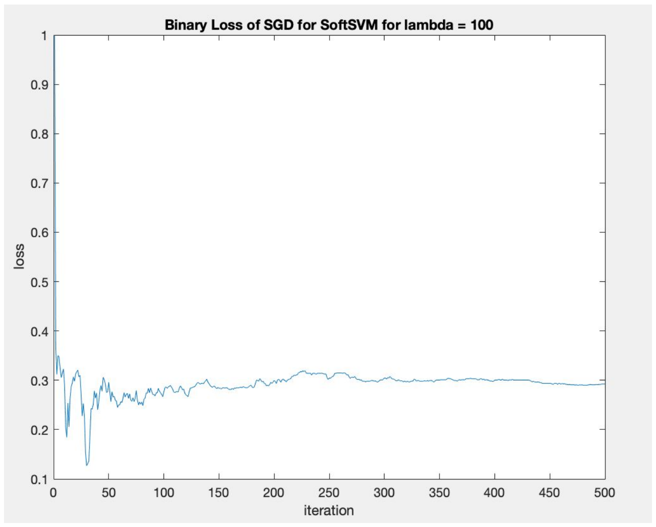
The magnitude of each update for the logistic and exponential gradients depends highly on t<w,x> as the functions have changing slopes. The quadratic gradient, being a straight line, does not depend on t<w,x> as it consistently updates the same amount due to the consistent slope.

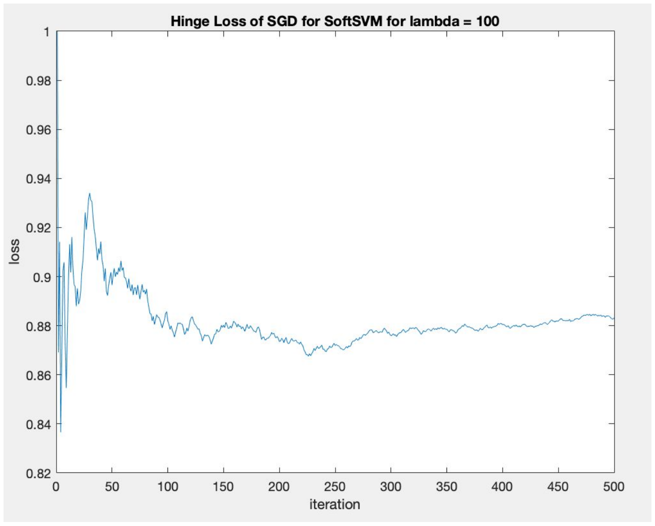
SoftSVM Optimization

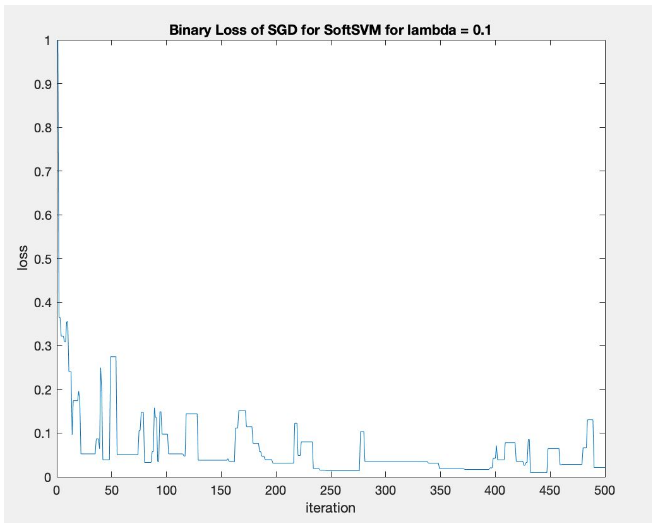
2a)

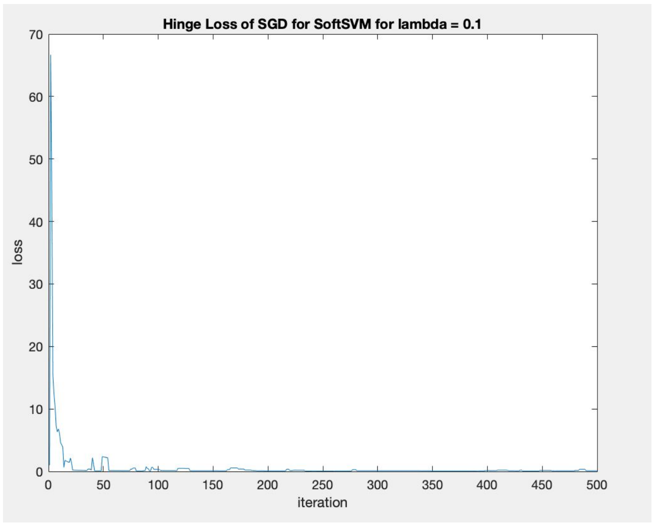
```
input = load("data_banknote_authentication.txt");
data = input;
data(:,5) = 1;
labels = input(:,5);
labels(labels(:)==0) = -1;

theta = zeros(1,5);
w = zeros(1,5);
T = 500;
lambda = .001;
i = 1;
b_loss = zeros(1,T);
h_loss = zeros(1,T);
for j = 1:T
    w = (1/(lambda * j)) * theta;
    i = ceil(rand * numel(labels));
    update = labels(i) * dot(w, data(i,:));
    if update < 1
        theta = theta + labels(i) * data(i,:);
    end
    b_loss(j) = binary_loss(w,data,labels);
    h_loss(j) = hinge_loss(w,data,labels);
end
```

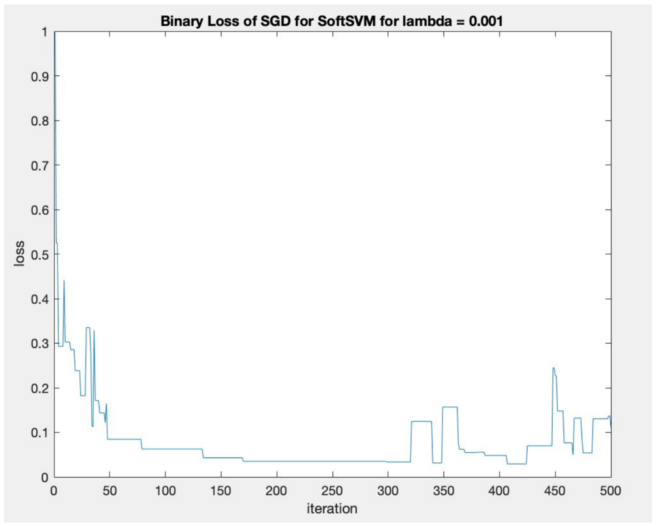
2b)

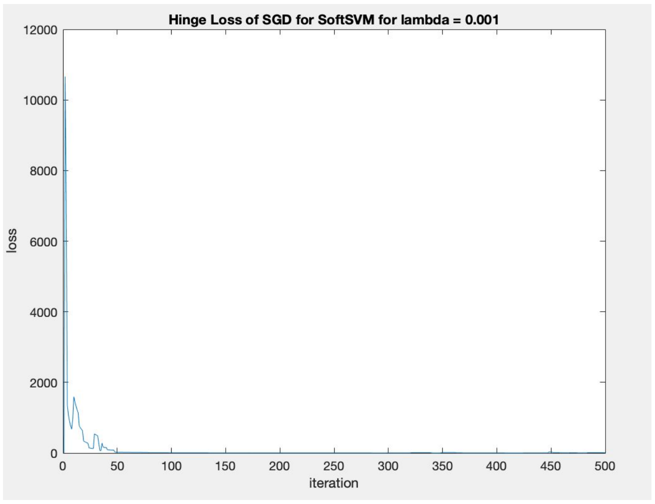












2c)

The curves are approximately monotonic, with the higher values of lambda showing more consistent descent of error-rates as the iteration counts go up. This is also more apparent due to the high degree of error present in the early iterations of the low valued lambda such as 0.001.

From the shown plots it can be seen that as the lambda value decreases, the variability of binary loss from one iteration to the next is much higher. This variability follows from the updating of the weight vector:

$$w^j = \frac{1}{\lambda j} * \Theta^j$$

With a small value of lambda, the amount of change of each update gets very large, causing more variability in the predictor weights.

To train a linear predictor of minimal loss I would use a high lambda value paired with a greater than 500 iterations. This combination would utilize small updates due to the large lambda over many iterations. Then take an average over the last 5%-10% of the generated weight vectors to smooth out any variability of the randomly selected data point for updating the weight vector.

2d)

Using Perceptron Algorithm

Binary loss for each of the “one-vs-all” classifiers using perceptron algorithm without normalized weight vectors:

loss_w1 = 0.3333, loss_w2 = 0.1429, loss_w3 = 0.3524

Binary loss for argmax <wi,x>: 0.4619

Binary loss for each of the “one-vs-all” classifiers using perceptron algorithm with normalized weight vectors:

loss_w1 = 0.3333, loss_w2 = 0.1429, loss_w3 = 0.3524

Binary loss for argmax <wi,x>: 0.4619

Normalizing the weight vector determined using the perceptron algorithm does not affect the binary loss when selecting the predictor label based on the max of the dot products of the three weight vectors and a given point. This follows from normalizing the weight vector not changing the hyperplane separating the one-vs-all classifiers trained. Normalizing the weight vectors does change the actual value of the dot product, but not the relationship between the three dot product values for the three one-vs-all classifiers.

With no change in loss with or without normalization both are valid for a multi-class predictor. If, instead, I had used the SGD algorithm there would be a benefit to using normalized values (both weight and data) to produce more consistent and smaller steps of the algorithm during gradient descent.

```
function weight = perceptron(inputs, labels)
[rows,cols] = size(inputs);
updates = 0;
error = -1;
weight = zeros(1,cols);
j = 1;
while j<=100 && error == -1
    %assume no more training required
    error = 0;
    for i=1:rows
        sign = labels(i) * dot(weight, inputs(i,:));
        if sign <= 0
            weight = weight + labels(i) * inputs(i,:);
            updates = updates + 1;
            %more training is required
            error = -1;
        end
    end
    j = j+1;
end
weight = weight ./ norm(weight);
end
```