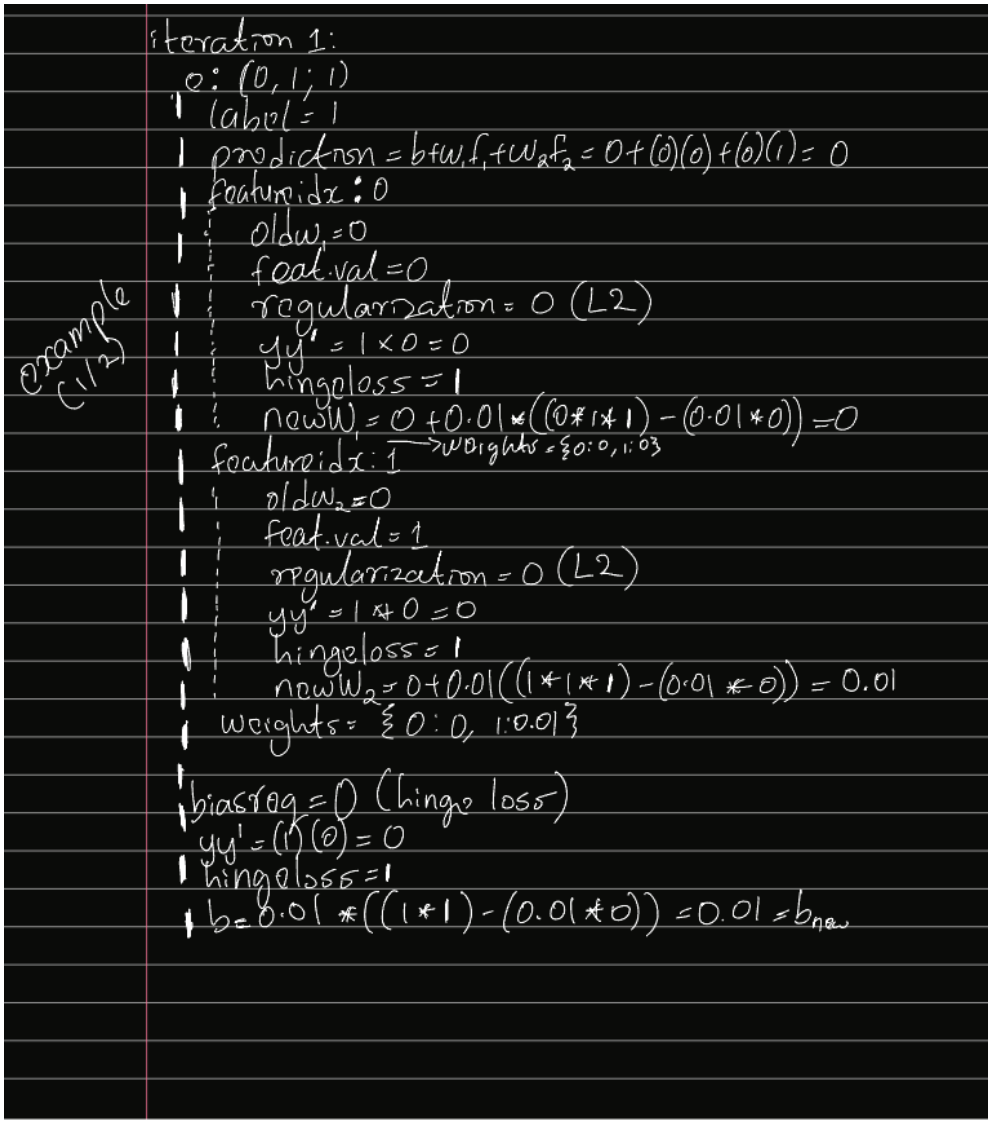
Algorithm Correctness

This is the testing data we used to check our work:

f1	f2	label
0	1	1
1	0	-1

We worked through our implementation through a single iteration and this is what we hand-wrote. For the first example's (0, 1; 1) iteration, we found:



The corresponding print statements added into our implementation yielded the same results as seen below:

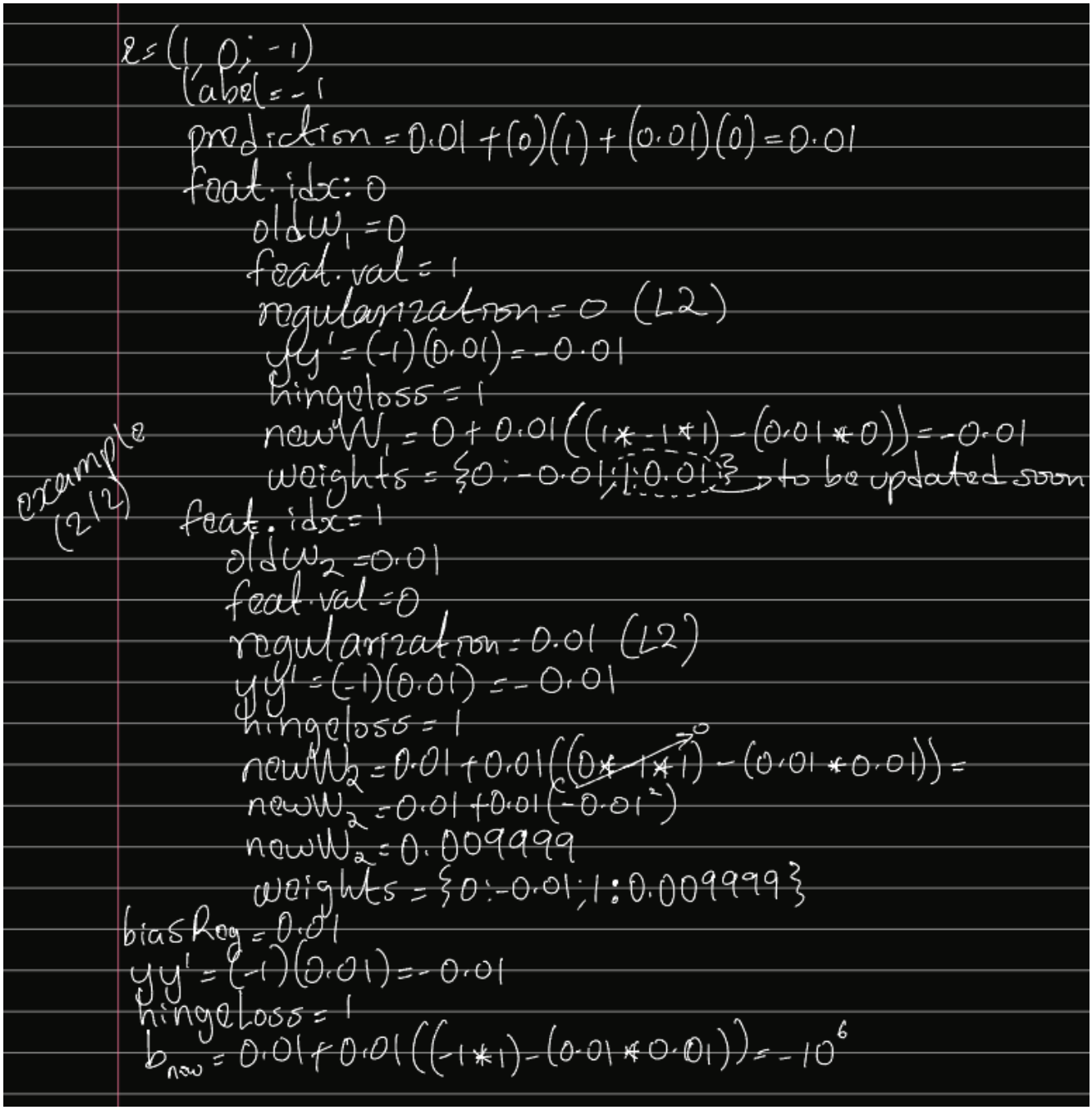
```
for (int it = 0; it < iterations; it++) {
    // Collections.shuffle(training);

    for (Example e : training) {
        // print out the current example
        System.out.println("The example we're looking at " + e);

        double label = e.getLabel();
        double prediction = getDistanceFromHyperplane(e, weights, b);
        System.out.println("The prediction gotten is " + prediction);

        for (Integer featureIndex : e.getFeatureSet()) {
            double oldWeight = weights.get(featureIndex);
            double featureValue = e.getFeature(featureIndex);
            double newWeight = 0;
            double regularization = 0;

            if (regularizationType == L1_REGULARIZATION) {
                regularization = Math.signum(oldWeight);
            } else if (regularizationType == L2_REGULARIZATION) {
                regularization = oldWeight;
            }

            if (lossType == EXPONENTIAL_LOSS) {
                newWeight = oldWeight
                        + eta * (featureValue * label * Math.exp(-label * prediction)
                                - lambda * regularization);
            } else if (lossType == HINGE_LOSS) {
                double yyPrime = label * prediction;
                double hingeLoss = yyPrime < 1 ? 1 : 0; // c = 1[yy' < 1]
                newWeight = oldWeight + eta * ((featureValue * label * hingeLoss) - (lambda * regularization));
                System.out.println("The new weight " + (featureIndex + 1) + " is " + newWeight);
            }
            weights.put(featureIndex, newWeight);
            // print out the weights      You, 1 hour ago • Testing the class on mock example …
            System.out.println("The weights after the update " + weights);
        }

        double biasRegularization = 0;
        if (regularizationType == L1_REGULARIZATION) {
            biasRegularization = Math.signum(b); // r = sign(b) for L1
        } else if (regularizationType == L2_REGULARIZATION) {
            biasRegularization = b; // r = b for L2
        }

        if (lossType == EXPONENTIAL_LOSS) {
            b += eta * (label * Math.exp(-label * prediction) - lambda * biasRegularization);
        } else if (lossType == HINGE_LOSS) {
            double yyPrime = label * prediction;
            double hingeLoss = yyPrime < 1 ? 1 : 0;
            b += eta * ((label * hingeLoss) - (lambda * biasRegularization));
        }
        // print out the bias
        System.out.println("The bias after the update " + b);
    }
```

The print statements yielded

```
The example we're looking at 1.0 0:0 1:1
The prediction gotten is 0.0
The new weight 1 0.0
The weights after the update {0=0.0, 1=0.0}
The new weight 2 0.01
The weights after the update {0=0.0, 1=0.01}
The bias after the update 0.01
```

This is exactly what we calculated above.
==========

For the second example's (1, 0, -1) iteration, we found:



Compare this against the run for this example (the final line includes the final weight vector and bias values):

```
The example we're looking at -1.0 0:1 1:0
The prediction gotten is 0.01
The new weight 1 is -0.01
The weights after the update {0=-0.01, 1=0.01}
The new weight 2 is 0.01
The weights after the update {0=-0.01, 1=0.01}
The bias after the update 0.0
0:-0.01 1:0.01 b:0.0
```

Here, we observe noteworthy developments. Weight 1 values are nearly identical at approximately -0.01. However, weight 2 values exhibit a slight disparity: our calculation yields 0.009999, while the code outputs 0.01. This marginal difference is also evident in the bias value, with the code outputting 0 and our result being -10^6. We suspect that the variation arises from a type-related issue in handling double values, potentially involving truncation or rounding during computation. This hypothesis is substantiated by the fact that our bias is very close to 0, and weight 2 is rounded to 0.01, indicating a loss of precision. In summary, our implementation appears to be correct, given the remarkably close agreement between our results and the code output.
==========

Next, we're going to present an argument about how our implementation follows the pseudocode given in lecture 10 to the t.

Our code (snippet):

```java
for (int it = 0; it < iterations; it++) {
    Collections.shuffle(training);

    for (Example e : training) {
        // print out the current example
        System.out.println("The example we're looking at " + e);
```

The pseudocode:

repeat until convergence (or for some # of iterations):

randomly shuffle the training data

for each training example (x_i, y_i):

Note: We carry out gradient descent for some number of iterations, shuffle the data, and then carry out the gradient descent on each example.

Our code (snippet):

```
double prediction = getDistanceFromHyperplane(e, weights, b);
System.out.println("The prediction gotten is " + prediction);
for (Integer featureIndex : e.getFeatureSet()) {
    double oldWeight = weights.get(featureIndex);
    double featureValue = e.getFeature(featureIndex);
    double newWeight = 0;
    double regularization = 0;      munir-v, 6 days ago • add regularization to train method …
    if (regularizationType == L1_REGULARIZATION) {
        regularization = Math.signum(oldWeight);
    } else if (regularizationType == L2_REGULARIZATION) {
        regularization = oldWeight;
    }

    if (lossType == EXPONENTIAL_LOSS) {
        newWeight = oldWeight
                + eta * (featureValue * label * Math.exp(-label * prediction)
                        - lambda * regularization);
    } else if (lossType == HINGE_LOSS) {
        double yyPrime = label * prediction;
        double hingeLoss = yyPrime < 1 ? 1 : 0; // c = 1[yy' < 1]
        newWeight = oldWeight + eta * ((featureValue * label * hingeLoss) - (lambda * regularization));
        System.out.println("The new weight " + (featureIndex + 1) + " is " + newWeight);
    }
    weights.put(featureIndex, newWeight);
    // print out the weights
    System.out.println("The weights after the update " + weights);
}
```

The pseudocode:

for each weight:

$$w_j = w_j + \eta(y_i x_{ij} c - \lambda r)$$

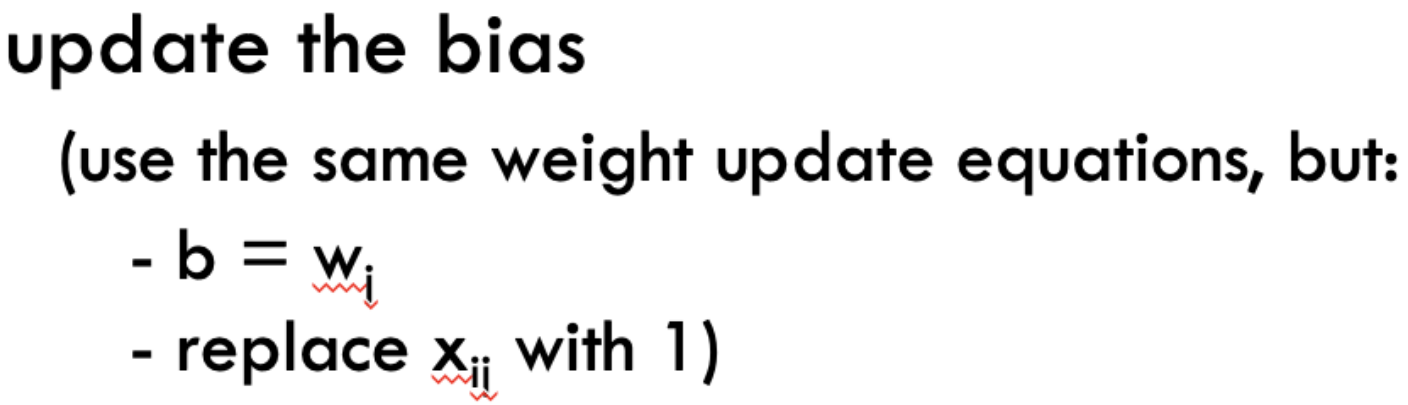
In this context, we address individual weights corresponding to feature indices. This involves iterating through each feature index for every example. Our code branches based on the regularization and loss types, encapsulated in the pseudocode equation as follows. The value of "r" depends on the chosen regularization type. For instance, in the case of using Hinge Loss with L2 regularization, "r" corresponds to the previous saved weight (w_j), as implemented in our code. The loss type determines the value of "c" in the equation. In the case of Hinge Loss, "c" is derived from the indicator function: c = 1[yy' < 1], assigning 1 to yy' values less than 1 and 0 otherwise. Here, "y" represents the example's label, and "y'" is the prediction, which equals the distance from the hyperplane: b + w1f1 + w2f2. This indicator function is elegantly captured by the ternary equation.

Finally, we have the bias update. We treat the bias update as we would a weight update but it has to be done once all (or both, in our case) weights have been updated.

Our code (snippet):

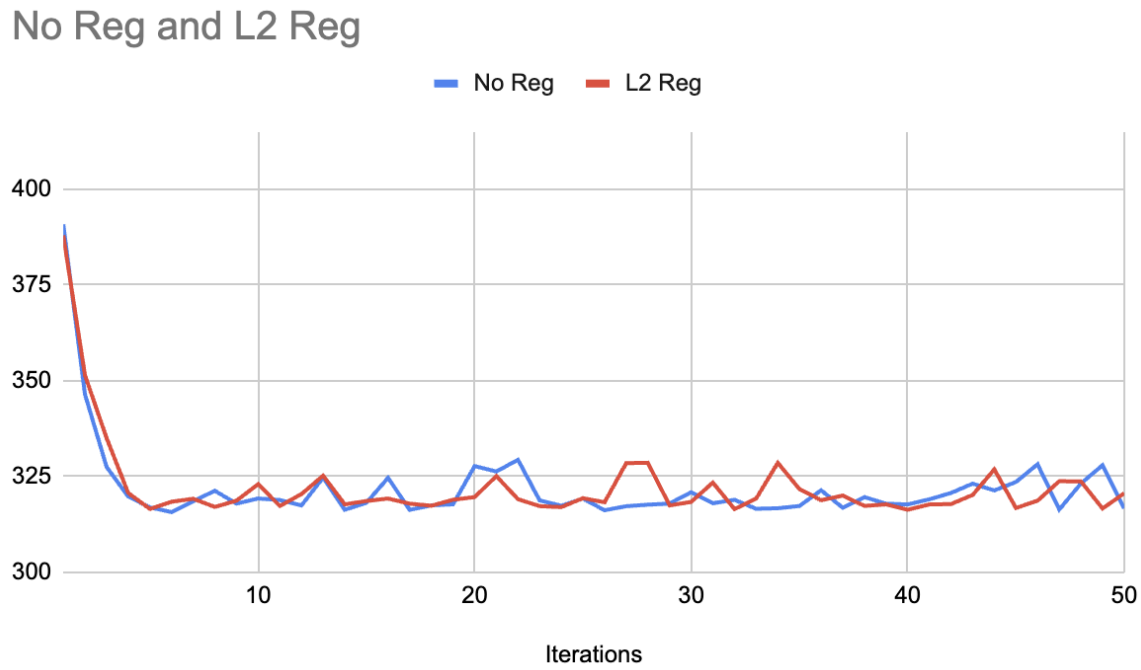
```
    double biasRegularization = 0;
    if (regularizationType == L1_REGULARIZATION) {
        biasRegularization = Math.signum(b); // r = sign(b) for L1
    } else if (regularizationType == L2_REGULARIZATION) {
        biasRegularization = b; // r = b for L2
    }

    if (lossType == EXPONENTIAL_LOSS) {
        b += eta * (label * Math.exp(-label * prediction) - lambda * biasRegularization);
    } else if (lossType == HINGE_LOSS) {
        double yyPrime = label * prediction;
        double hingeLoss = yyPrime < 1 ? 1 : 0;
        b += eta * ((label * hingeLoss) - (lambda * biasRegularization));
    }
    // print out the bias
    System.out.println("The bias after the update " + b);
}

double currentLoss = getTotalLoss(training);
losses.add(currentLoss);
```

The pseudocode:



The weight update equation remains consistent, with the feature value replaced by 1, as we have done. The specific bias update is determined by the chosen regularization and loss types. It's worth noting that the regularization value for the bias (b_r, if you prefer) is essentially equivalent to "b," as "r" previously represented "w_j" in the weight update equations, in accordance with the pseudocode. Consequently, we firmly believe that we have faithfully adhered to the pseudocode and that our implementation is indeed correct and acceptable.

Experimentation



The blue line shows the graph of the hinge loss function with no regularization and the default parameters. We see a quick drop in loss since the model can overfit to the training data. The red line shows the graph of the hinge loss function with L2 regularization. We see a very similar graph. This would change if we included regularization in the calculation of our loss value (but Dr. Dave said we didn't need to).

For our additional experiment, we wrote a program to find the optimal lambda/eta values for the exponential loss function with default parameters. The program trained the model on a combination of lambda and eta values from the following list: { 0, 0.001, 0.002, 0.003, 0.004, 0.005, 0.01 }.

Our best lambda and eta values (i.e. the values that led to the lowest loss) are below:
Optimized lambda: 0.003
Optimized eta: 0.01