Skidmore College

CS 316: Machine Learning
Fall 2023: Lab 3
Linear Two-Class Classification

Travis McVoy

December 16, 2023

Abstract

In this report we will explore the basics of linear two-class classification by manually implementing a perceptron learning algorithm and gradient descent for various loss functions. We'll also get some practice using the SciPy package when we implement a classifier to find a halfspace for a linear program. Finally, we will test our implementations on linearly-separable data that we will randomly generate ourselves. The performance of each learning algorithm can be found in the results seaction, whereas all other sections highlight various details about the functions necessary for the progam.

1 Generating Data

One need not generate data prior to implementing any of the learning algorithms. However, generating data first will enable us to test each algorithm after implementing it, as opposed to testing all algorithms at once. In the end, it probably doesn't actually reduce the number of bugs to do data first, algorithms second, debugging one algorithm at a time is certainly more pleasant.

To generate data, we will implement two functions: `initvec()` and `genData()`. The `initvec()` function is a helper function that will return a random vector (array, if you prefer) of dimension `dim`, which is passed as a parameter. The `genData()` generates our $\mathbf{x}$ instances and their corresponding y labels based on a weight vector $\mathbf{w}$. I chose to store my

bias in the weight vector, as that makes the code a little cleaner in the long run; instead of writing something along the lines of `torch.dot(instance, weight) + bias` we can just write `torch.dot(instance, weight)`. To generate a random weight vector (including bias), I used a method from NumPy. Now that we have a weight vector, we can generate the training and testing data.

To ensure the data is linearly separable, for each $\mathbf{x}$ instance we create, we manually assign a y label that corresponds to the value of the dot product between the weight vector and the instance vector. The actual value of the label will change depending on what type of learning algorithm we use (soft max uses -1 instead of 0), but generally we assign a label in the following manner: if $\mathbf{x}^T\mathbf{w} > 0$ then the corresponding y label is $+$, otherwise $-$ where $+, -$ take on the appropriate binary classification values for each learning algorithm.

There are a few final details about the `genData()` function that should not be overlooked. We are doing a binary classification, so one might expect our $\mathbf{x}$ instances to be two dimensional, but they are not. Because we are storing the bias as the last element of the weight vector, we need to append a 1 onto each $\mathbf{x}$ instance to ensure the dot product is defined. Lastly, I have written the `genData()` function such that it will generate both our training data and our testing data all in one call. Ultimately, this is done by doubling the number of instances we wish to generate (1 train instance and 1 test instance across all samples) and then using a train/test split method from the sklearn library.

By calling the train/test split method, we return a tuple that contains four elements: an array of $\mathbf{x}$ training instances, an array of $\mathbf{x}$ testing instances, an array of y training labels, and an array of y testing labels. The idea of the train/test split is this: since we are generating our data, we know the exact weight vector that our learning algorithm should find (the random weight vector we chose initially). However, we can't just assume the algorithm works perfectly, so we generate test data from the same weight vector. Then, once our algorithm has finished training, we test the weight vector the algorithm returned on our testing data. If it correctly classifies everything, that's good, if it doesn't, that's generally bad (soft max is a special case). With that, we are ready to start implementing.

2 Logisitic Regression

We will use gradient descent to implement logisitc regression with one of the following three loss functions: least squares, cross entropy, or soft max. To reduce debugging pain, we start by verifying that each loss function works by choosing a few simple input vectors and calculating the loss by hand in desmos.

2.1 Least Squares

The least squares loss function for logisitic regression is defined as

$$L_S(\mathbf{w}) = \frac{1}{n}\sum_{k=1}^{n}\Big(\sigma(t\mathbf{x}_k^T\mathbf{w}) - y_k\Big)^2 \tag{1}$$

where σ is continuous approximation of the step function and t is a hyper parameter (the larger the t value, the steeper the approximation). To check our values in desmos, we will need the formula for the sigma function; it is given by

$$\sigma(tx) = \frac{1}{1+e^{-tx}}.$$

To check our values, we choose very simple (meaningless) instance vectors:

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{x}_2 = \begin{bmatrix} -2 \\ 3 \end{bmatrix}, \mathbf{x}_3 = \begin{bmatrix} 3 \\ -4 \end{bmatrix}$$

and an equally meaningless weight vector

$$\mathbf{w} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Finally, we choose $t = 1$, which again, is just to make manual computation fast. The dot product between an instance and a weight vector is now just adding up the elements in the vector, so our labels are

$$y_1 = 1, y_2 = 1, y_3 = 0.$$

When we plug everything into desmos, we get a loss value 0.0489693965679 which closesly matches the value that we got using our Python function (0.0489693917). To see the calculation in desmos, use this hyperlink.

2.2 Cross Entropy

We find cross entropy loss using

$$L_S(\mathbf{w}) = -\frac{1}{n}\sum_{k=1}^{n} y_k \ln\Big(\sigma(t\mathbf{x}_k^T\mathbf{w})\Big) + (1 - y_k)\ln\Big(1 - \sigma(t\mathbf{x}_k^T\mathbf{w})\Big). \tag{2}$$

It may not be entirely clear what each term is doing, so let's just quickly review. Like least squares, cross entropy uses 0,1 binary classification. Hence, when our label is zero, only the second term gets counted towards the loss (and vice versa for 1). When y_k is 0, our dot product is less than or equal to zero. Since we are taking logs, this could be a probelm as the domain of logarithmic functions must be positive. This domain hiccup explains the $1 - \sigma$ term in the second logarithm. The logs themselves exist to rescale the loss function value. Without them, the loss can blow up faster than we want it to (remember in the derivation for cross entropy, we started by considering the inverse value of the sigma function, that is, $1/\sigma$, which blows up when $\sigma \approx 0$).

Now that we remember what cross entropy is doing, we can check that our Python function works by again using the same values and plugging everything into desmos. After a little experimentation, I found that PyTorch is using the natural logarithm, as my values matched when I used the natural log in desmos. The PyTorch value and desmos value were repsectively

$$0.22503690421581268$$
$$0.22503690887$$

which is close enough for this report. Again, the desmos computation can be found with this hyperlink.

2.3 Soft Max

Soft max is slightly different than the previous two losses we've considered. Unlike least squares and cross entropy, soft max uses -1,1 classification whereas least squares and cross entropy use 0,1. The soft max loss is found with

$$L_S(\mathbf{w}) = \sum_{k=1}^{n} \ln\left(1 + e^{-ty_k \mathbf{x}_k^T \mathbf{w}}\right) \tag{3}$$

where t is a hyper parameter like before. When we check with desmos, our values match. In fact, they are the same values that we found for cross entropy. This is not a coincidence. Cross entropy and soft max are equivalent, but soft max allows us to use different labels

2.4 Gradient Descent

Now that we have implemented our loss functions, we can implement logistic regression via gradient descent. We do this by calculating the loss, taking the gradient, traveling in the opposite direction of the gradient and updating our weight accordingly, and then repeating until some termination criterion has been met. I have two conditions that allow the gradient descent to stop. The first condition is a maximum number of iterations, which I have chosen to be 2000. The number 2000 was chosen after experimenting with my test functions. If we iterate 2000 times, that takes my machine roughly 30 seconds. A runtime of 30 seconds per sample corresponds to about several hours of waiting when collecting data for even a small (100 samples) sample space. Hence, 2000 iterations seemed like the best I could reasonably do. Alternatively, we can use the value of the loss to determine if we should terminate. Namely, if the loss is quite small, we can probably stop. It isn't quite clear how small is small enough. I'd argue it depends on the desired accuracy. Since this is just an experiment, I allowed my algorithm to terminat once a loss smaller than 0.2 was achieved. Doing so sped up the time I had to wait, but at the cost of accuracy.

3 Perceptron Learning Algorithm

Since our data is linearly seperable, we can tell the perceptron to run until it has not miss-classified any of the training instances. At every iteration, we update the weight by adding the incorrectly classified istance scaled by it's label. Doing so will achieve the following change:

x_{ij}	$y_i = -1$ but output is 1	$y_i = +1$ but output is -1
$+$	decrease w_j	increase w_j
0	no change	no change
$-$	increase w_j	decrease w_j

4 Linear Programmming with ScyPy

Consider a rather simple linear program givien by

- maximize $10s + 2a - 4p$

- subject to:

$$
\begin{aligned}
s + a + p &\leq 16 \\
s &\geq 2 \\
a &\geq 2 \\
p &\geq 2 \\
s &\leq 6 \\
a &\leq 10
\end{aligned}
$$

We can use SciPy to find an optimal solution with the following code (obtained from lecture notes):

```
from scipy.optimize import linprog

# maximize 10s + 2a - 4p
# minimize -10s - 2s + 4p
objective = [-10, -2, 4]
# s + a + p <= 16,
# s >= 2 -> -s <= -2
# a >= 2 -> -a <= -2
# p >= 2 -> -p <= -2
# s <= 6
# a <= 9
inequalities = [[1, 1, 1], [-1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 0, 0], [0, 1, 0]]
inequality_rhs = [16, -2, -2, -2, 6, 9]
result = linprog(objective, inequalities, inequality_rhs)
print('result = ' + str(result))
print('optimal objective value = ', result.fun)
print('s, a, p = ',  result.x)
```

If we want to find a half space using linear programming, we do a similar process only our object function is 0 and we need to generate our coefficients using a loop. That is, we let our constraints be

$$y_i(\mathbf{x}_i^T \mathbf{w}_i) \geq 1$$

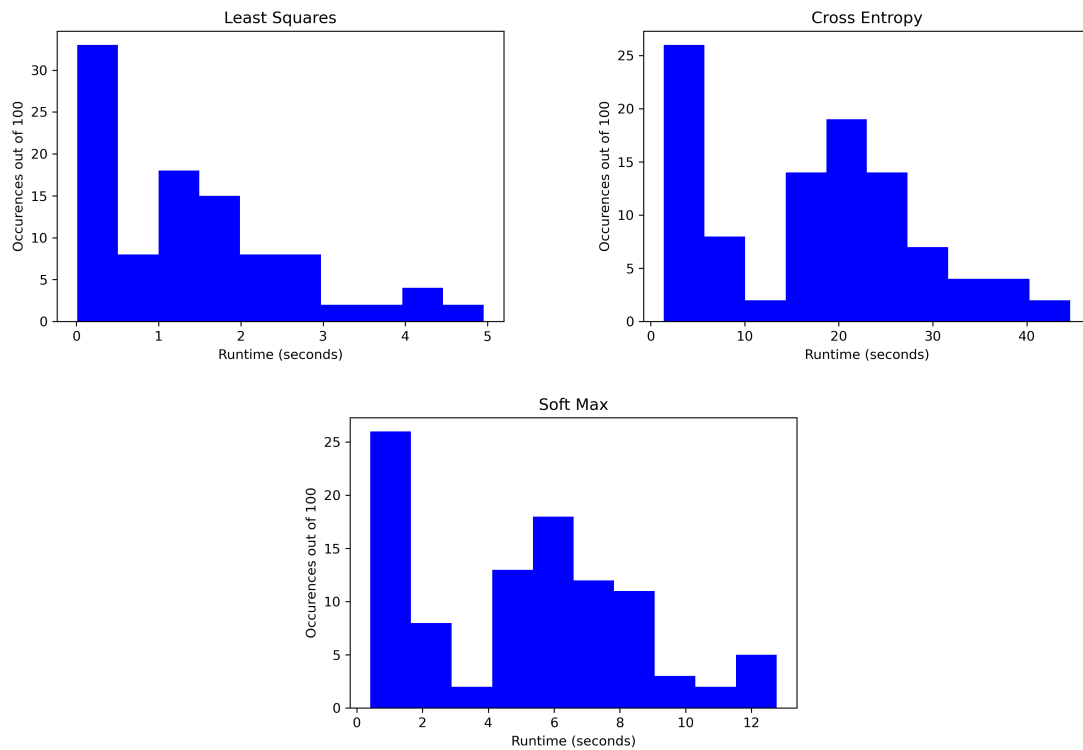
for all i.

5 Results

To test our learning algorithms, we can generate 100 samples of 200 instances each (100 training instances and 100 testing instances) and run each algorithm on each sample, collecting data as we do so. In the following sections, we analyze the runtime, number of iterations, training accuracy, and testing accuracy of each algorithm.

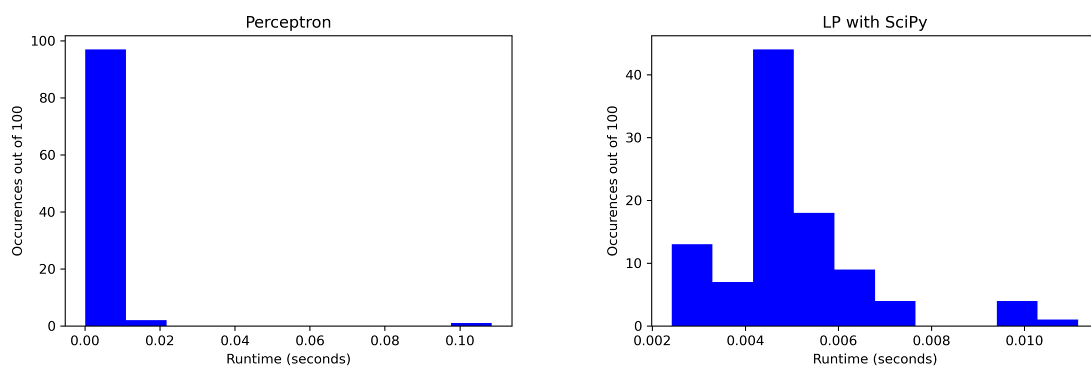
5.1 Runtime

Below are the histograms for the runtime data for the gradient descent algorithms:



Notice the discrepancy between cross entropy and soft max, despite the fact that the two loss functions compute the same values. The difference is likely due to the extrea terms in 2. Notice also that least squares ran rather quickly. It's possible that the least squares algorithm reaches a smaller loss value faster. What isn't clear is why a smaller loss value does not necessarily imply better accuracy. While least squares certainly ran faster than the other gradient descent algorithms, when we examine accuracy we will find that the short runtime would seem to come at a cost.

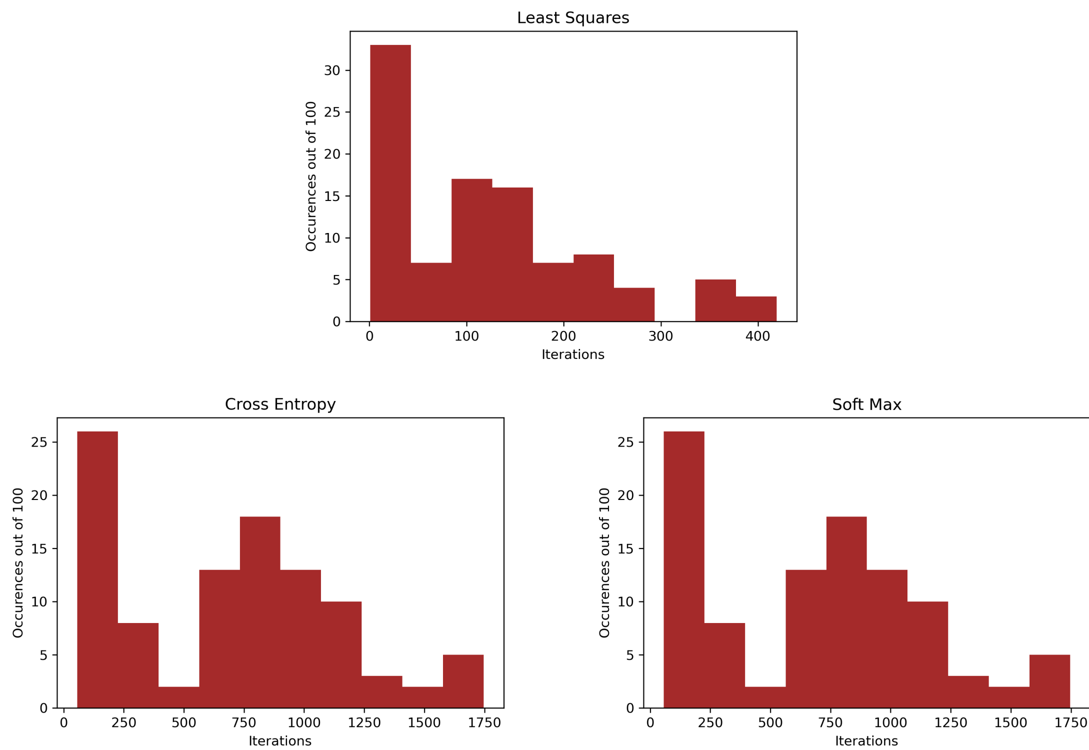
Let us turn our attention to the perceptron and linear programming algorithms:



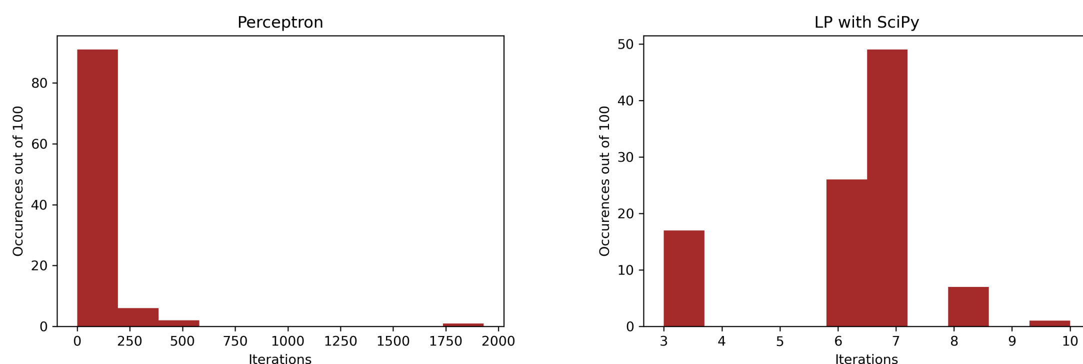
It should be clear from inspection that the perceptron had at least one outlier. Unfortunately, this removes a lot of the information from our histogram, as it disrupts the scale and places the vast majority of all cases in a single bin. When I clean up this lab for a portfolio, I want to figure out how to use some sort of statistics package to identify outliers and give me a better plot.

5.2 Iterations

We start with the iterations for the gradient descent algorithms:



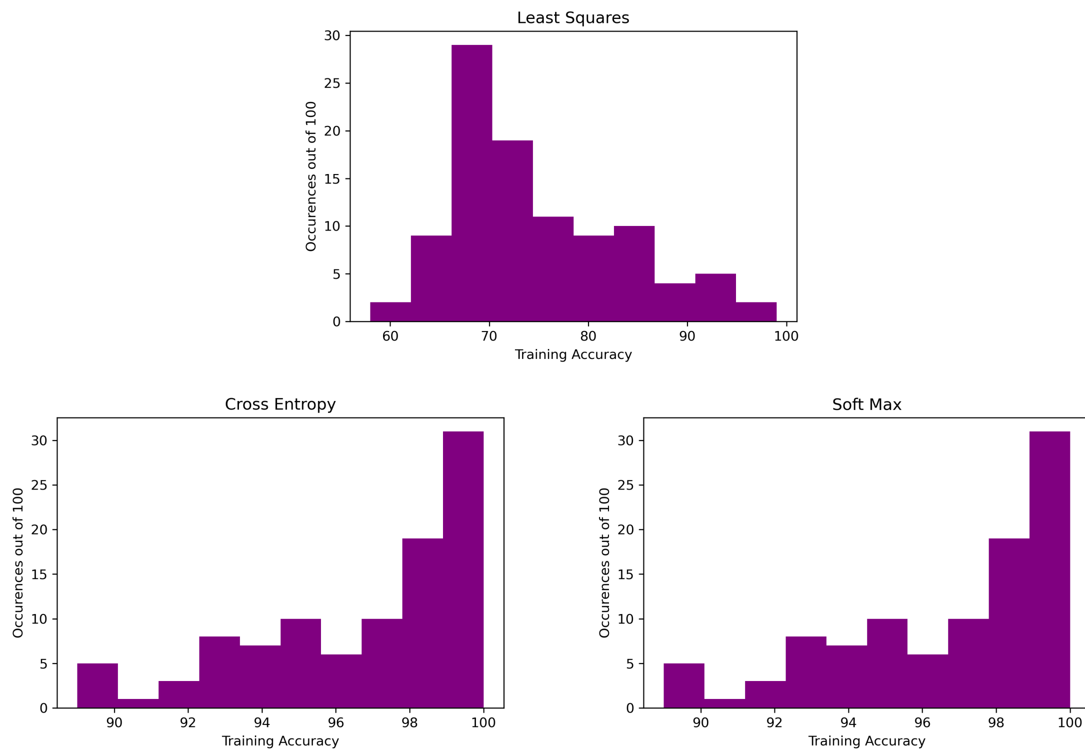
We can see that cross entropy and softmax appear to have the same distribution of iterations, which matches what we expect given they comput the same values. We see also that the least squares never got close to the iteration limit of 2000, verifying our intuition about the running time. We chose a higher loss cutoff (0.2) than is ideal, so it makes sense that our iterations and accuracy are not as high as they could be. We see below that the perceptron and linear programming are both much faster than gradient descent, not just in runtime, but in iterations as well:



That pesky outlier is still destroying our scale, but not all is lost. The vast majority of the perceptron iterations are fewer than 250, but it isn't clear what the distribution is among those values. If I have more time, I'll do another test only the perceptron so as to get more informative data.

5.3 Training Accuracy

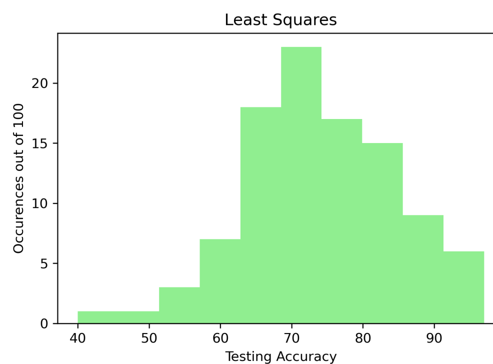
The training accuracy of both the perceptron and the linear progamming algorithms was 100% for both, which is what we would expect. Hence, we focus only on the gradient descent algorithms:

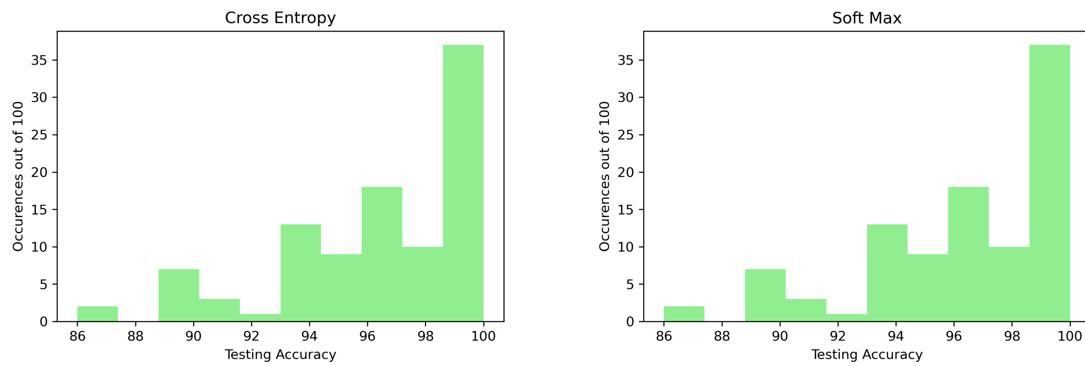


Again, cross entropy and soft max compute the same values, so it makes sense that their histograms appear identical. As mentioned earlier, least squares is not as accurate as cross entropy or soft max, and to a staggering degree. Least squares was less than 80% accurate for the majority of the training samples, whereas cross entropy and soft max don't appear to have a single instance that low.

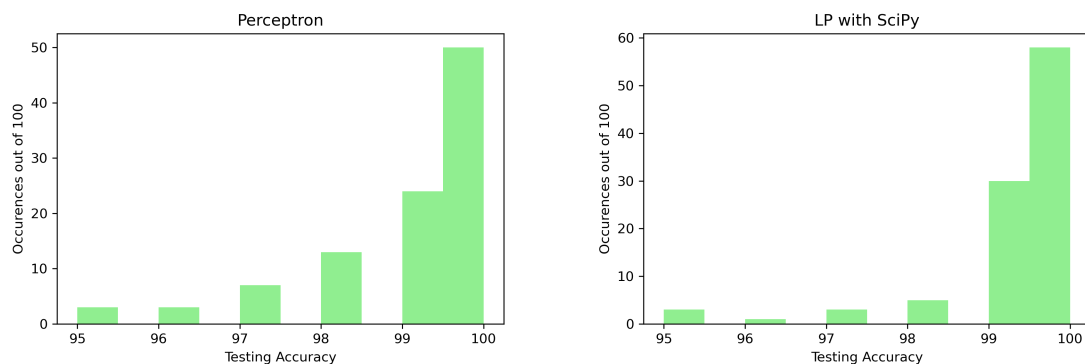
5.4 Testing Accuracy

We start with gradient descent. Like before, cross entropy and sot max appear to be identitical, which is good because they should be. For the most part, it seems that cross entropy and soft max generalized well, as the majority of the samples had an accuracy of 94% or higher, which matches the results from training. However, unlike the training data, there are a few samples with slightly lower accuracy rates (less than 90 %). Least squares didn't generalize very well at all, which shouldn't be a surprise at this point.





The perceptron and linear programming algorithms did quite well with the majority of samples having an accuracy of 98% or higher:



6 Conclusion

If we know the data is linearly separable, linear programming or perceptrons are the clear choice. Linear programming with SciPy will definitely be faster (see the iterations histograms) but it requires that the user can use the SciPy package. SciPy package isn't too hard to use, but I found it wasn't particularly easy to debug. Since the majority of the code ins't visible to the user, a slight error can be difficult to find. I forgot to multiple by negative one when initially implementing, and that ruined my results. The perceptron, on the other hand, was incredibly easy to implement and provides similar performance with respect to accuracy. There are a few questions that we may have. In particular, why do certain algorithms run faster? In the case of the perceptron, it could be that there is an instance that is close to, but not exactly zero. Such an instance will barely update the perceptron, and could then continue to barely update after many iterations. It isn't entirely clear to me why least squares ran so much faster than cross entropy or soft max. It could be the case that least squares starts with a lower loss to begin with, and therefore reaches the cutoff with fewer iterations. I don't think it is the case that the gradient of the least squares loss is higher. Quite the contrary, I found that I could crank up my step size to about 50 before the least squares loss became unreliable (the same results were not even remotely achievable with soft max and cross entropy).