CS5350 Assignment 2

Corey Schulz

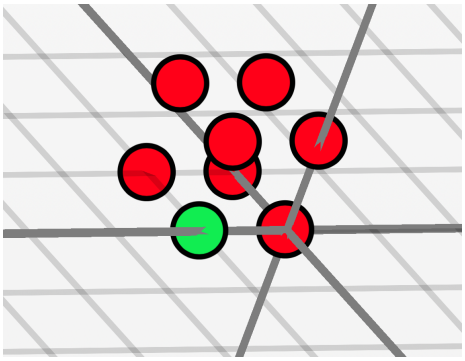
September 21, 2019

1 Linear Classifiers and Boolean Functions

1.

$$\neg x_1 \wedge x_2 \wedge \neg x_3$$

Plotting the points for this equation in a 3D space results in the following:



Visually, we can see that it is linearly separable, as a plane *can* segment it off from the rest of the points.

$$(1 - x_1) + x_2 + (1 - x_3) \geq 3$$
$$-x_1 + x_2 - x_3 + 2 - 3 \geq 0$$
$$-x_1 + x_2 - x_3 - 1 \geq 0$$

Input	Outcome	$-x_1 + x_2 - x_3 - 1$	Sign
[0, 0, 0]	-	-1	0
[0, 0, 1]	-	-2	0
[0, 1, 0]	+	0	1
[0, 1, 1]	-	-1	0
[1, 0, 0]	-	-2	0
[1, 0, 1]	-	-3	0
[1, 1, 0]	-	-1	0
[1, 1, 1]	-	-2	0

Then, finally: $\mathbf{w} = [-1, 1, -1]$; $\mathbf{b}$ = -1

2. $$(x_1 \overline{\oplus} x_2) \oplus x_3$$

   ...Cannot be linearly separated because, by definition, the XOR function ($\oplus$) is not itself linearly separable.

3. $$x_1 \wedge (\neg x_2 \vee \neg x_3)$$

   We can determine this by the following function:

   $$sgn(2x_1 - x_2 - x_3 - 1)$$

   And making a table as before gives us:

Input	Outcome	$2x_1 - x_2 - x_3 - 1$	Sign
[0, 0, 0]	-	-1	0
[0, 0, 1]	-	-2	0
[0, 1, 0]	-	-2	0
[0, 1, 1]	-	-3	0
[1, 0, 0]	+	1	1
[1, 0, 1]	+	0	1
[1, 1, 0]	+	0	1
[1, 1, 1]	-	-1	0

   So, then: $\mathbf{w}$ = [2, -1, -1] ; $\mathbf{b}$ = -1

4. $$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee x_3$$

   ...Can be simplified to, or rewritten as, the following:

   $$(x_1 \oplus x_2) \vee x_3$$

   ...And since the equation contains an XOR function, by definition it is not, then, linearly separable.

5. $$\neg(x_1 \wedge \neg x_2) \vee x_3$$

   Can be rewritten as:

   $$\neg x_1 \vee x_2 \vee x_3$$

   Then,

   $$1 - x_1 + x_2 + x_3 \leq 0$$

   And to shift the plane down a bit:

   $$1 - x_1 + x_2 + x_3 - .5 \leq 0$$

   Then, we can get the following table:

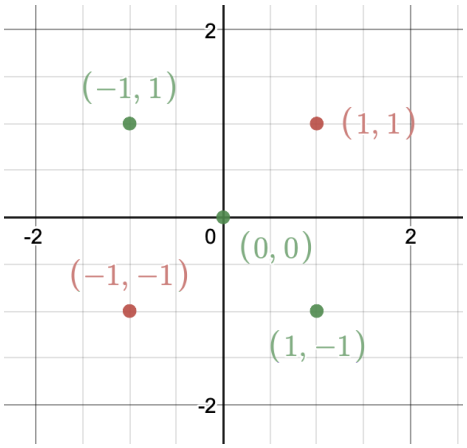
Input	Outcome	$-x_1 + x_2 + x_3 + .5$	Sign
[0, 0, 0]	+	.5	1
[0, 0, 1]	+	1.5	1
[0, 1, 0]	+	1.5	1
[0, 1, 1]	+	2.5	1
[1, 0, 0]	-	-.5	0
[1, 0, 1]	+	.5	1
[1, 1, 0]	+	.5	1
[1, 1, 1]	+	1.5	1

And then, finally: $\mathbf{w}$ = [-1, 1, 1] ; $\mathbf{b}$ = .5

2 Feature Transformations

Point	Outcome
[0, 0]	+
[1, 1]	-
[-1, 1]	+
[-1, -1]	-
[1, -1]	+

1. Plotting those points looks like the following, with + points represented by green and - points represented by red:



Regardless of how you draw the line, a single line cannot separate the red points from the green – the line would have to in some way bisect a green point, which is invalid. As such, the points are not linearly separable in $\mathbb{R}^2$.

2. The points, however, *can* be translated into a 3D space, in which case they *are* linearly separable. Merely shifting the negative points up, you

can put a plane in the middle of the shifted up points (negative points) and the non-shifted points (positive points). An equation that does this can be defined as follows:

$$\phi(x + y) = |x + y|$$

This transformation, which leaves the positive points (all along the line $y = -x$) at 0, while shifting up the negative points, would place the points in $\mathbb{R}^3$ where they are linearly separable. Applying function ϕ results in the updated points:

Point	Outcome
[0, 0, 0]	+
[1, 1, 2]	-
[-1, 1, 0]	+
[-1, -1, 2]	-
[1, -1, 0]	+

And again, these points *can* be linearly separated.

3. A weight vector and bias can be found as follows:

$$\phi(x, y) = |x + y|$$

$$ax_1 + bx_2 + cx_3 + b \leq 0$$

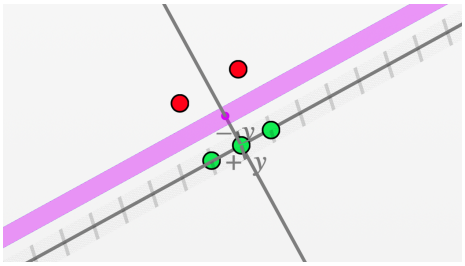
$$0x1 + 0x2 + cx_3 + b \leq 0$$

$$x_3 + b \leq 0$$

$$x_3 + 1 \leq 0$$

Weight vector $\mathbf{w} = [0, 0, 1]$

Bias $\mathbf{b} = 1$



...The plane intersects the z distance between the two points, and they are as such linearly separated.

3 Mistake Bound Model of Learning

1. (a) The concept class size can be determined as follows:

$$|C_{(n,l)}| = \binom{\sum_1^n 2^l \binom{n}{l}}{3}$$

For 1 to n, each variable l can have 2 states, and then (n C l) possible variables, so multiply. Then, choose three.

(b) An algorithm for this concept class that will make only a polynomial number of mistakes can be made can be defined as the following:

```
weights = zeros(size)
foreach given example (ex, label):
    predict pred = sgn(w^T * ex)
    if pred != label:
        w += label * ex
return weights
```

...Importantly, this algorithm is very similar to perceptron. As perceptron has polynomial time at its upper bound, and as that has been proven in class, this algorithm too applies. Its mistake bound is very clearly defined in a polynomial sense, given:

$$\sum_{l=1}^{n} 2^l * n^l$$

...Which is a large number that grows incredibly quickly, but still it's polynomial nonetheless. As such, the mistake bound is proven for this algorithm as the mistake bound is already proven for Perceptron, this draws the same concepts directly from Perceptron. QED.

2. **Extra Credit**

(a) An algorithm that is more efficient than the halving algorithm (which is itself space and time inefficient) could be defined as the following, something like the weighted majority algorithm

```
let w = [1, 1, ..., 1]; 0 <= beta <= 1
For t in range(T):
    receive x_t
    p_0 = x_t[x_t == 0].dot(w[x_t==0])
    p_1 = x_1[x_t == 1].dot(w[x_t==1])
    if p_1 >= p_0:
        predict 1
    else:
        predict 0
```

```
receive yt
for i in range(n):
    if x_t[i] != yt:
        w[i] *= beta
```

(b) In this case, the *absolute worst case* is when 'beta' is zero. The algorithm would function identically to the halving algorithm. When 'beta' is not zero, though, the algorithm uses less space and time by implementing the concept of weighting. As such, it can be proven that the *absolute worst case* is a polynomial number of mistakes, which is still fine given that it's the upper echelons of Big O. It's proven because the Halving Algorithm was stated in class as having polynomial time. QED.

4 The Perceptron Algorithm and its Variants

1. **Design Decisions**
   For this assignment, I used Python 3.7.4 to implement the four different types of Perceptron (Simple, Decaying, Average, Pocket). I represented the vectors as Numpy arrays so they can easily be added, multiplied, and dotted (something that the builtin Python array type doesn't support). Each Perceptron is its own class in the Perceptron.py file, and while they're all predominantly the same, each does have a different flavor, mainly in the Perceptron training algorithm itself. I did have to go back and modify each of the classes to have better data reporting, and I should have thought about that initially rather than doing it after the fact. There was no problem with parsing the data this time, as I found Perceptron to be easier to reason about the actual data we're given over ID3. I used predominantly arrays, then, to hold data for Perceptron across all types.

2. **Majority Baseline**
   The most frequent label in the training data set is **-1**, with a total of 799 occurrences out of 1581 labels . If it were to be used to predict the label on each entry in that set, it would have an accuracy of:

$$\frac{799}{1581} = .50537$$

   And for the test data, the majority label is **1**, with a total of 205 occurrences out of 396 labels. If it were to be used to predict the label on each entry in that set, it would have an accuracy of:

$$\frac{205}{396} = .51767$$

   ...Which, in both cases aren't good. And, if the test data used the majority label from the training data, it'd be below 50% accurate because -1 is not

the majority label in the training data. Not a very good way to predict these things.

3. **Word Weights**
   The overall best Perceptron variant is *Average Perceptron*, with *learning rate = 0.01* and training it with the training data produced an array of weights. I filtered out the 10 highest weights and got their indexes, then I used those indexes to access the words from the JSON object. The following words were the words with the highest classification weights:

Word	Weight
space	0.1720
nasa	0.1000
earth	0.0951
rocket	0.0909
orbit	0.0864
observatory	0.0834
international	0.0833
moon	0.0797
inflatable	0.0781
galaxy	0.0746

...Which makes sense for the words. A label of **1** is an article about space, and all of these words obviously relate strongly to space in one way or another (except for maybe *inflatable*, but that could be talking about spaceship parts?). Basically, if an article mentions one of these words, it's more likely to be classified as a space article, and that is reasonable.

Now, the words with negative weights:

Word	Weight
health	-0.1144
medical	-0.1121
med	-0.1065
normal	-0.0901
surgery	-0.0895
diet	-0.0782
msg	-0.0741
experience	-0.0733
medicine	-0.0725
told	-0.0704

...And the data also makes sense for these words, as a label **-1** corresponds to a medical article, and all the words in the above list would strongly correlate with medical terminology. It appears, then, that the classifiers work!

4. **Perceptron Variant Analysis**

1) **Simple Perceptron**

My cross-validation function automatically spits out the best performing hyperparameter for each Perceptron variant, and in the case of Simple Perceptron, it was **Learning Rate = 0.1**.

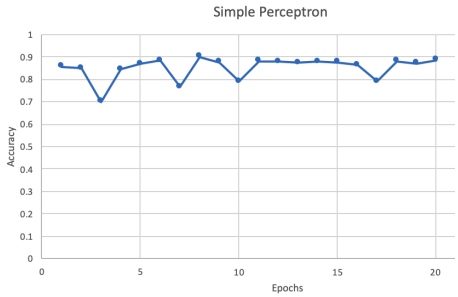
The cross validation accuracy for this hyperparameter was: **0.8710**.

Over 20 epochs, the number of updates that are made to the training set total: **4610 updates**.

After those 20 epochs are over, the accuracy on the training set is: **0.91524**

And the accuracy on the test set is: **0.8914**!

Finally, if accuracy data is collected after every epoch, then the learning curve for simple perceptron looks like this:



2) **Decaying Perceptron**

The best hyperparameter in the case of Decaying Perceptron **Learning Rate = 0.1**.

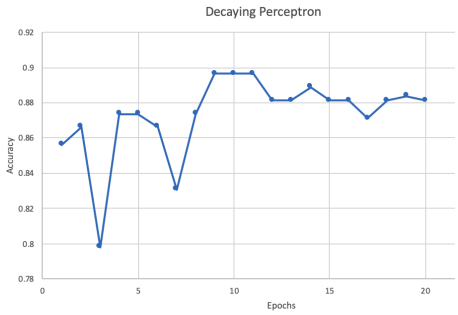
The cross validation accuracy for this hyperparameter was: **0.88675**.

Over 20 epochs, the number of updates that are made to the training set total: **3477 updates**.

After those 20 epochs are over, the accuracy on the training set is: **0.9184**

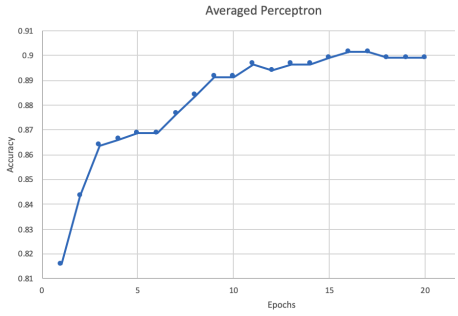
And the accuracy on the test set is: **0.9040**!

Finally, if accuracy data is collected after every epoch, then the learning curve for decaying perceptron looks like this:



Note the accuracy starts at a high number, and this graph is zoomed!

3) **Averaged Perceptron**
The best hyperparameter in the case of Averaged Perceptron **Learning Rate = 0.01**.
The cross validation accuracy for this hyperparameter was: **0.89119**.
Over 20 epochs, the number of updates that are made to the training set total: **5060 updates**.
After those 20 epochs are over, the accuracy on the training set is: **0.91714**
And the accuracy on the test set is: **0.9040**!
Finally, if accuracy data is collected after every epoch, then the learning curve for averaged perceptron looks like this:



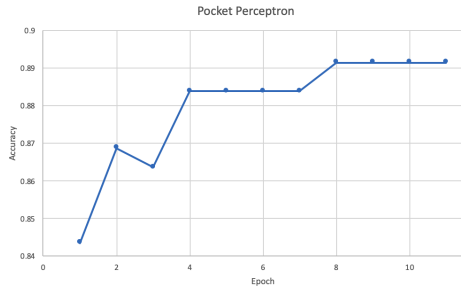
Note the accuracy starts at a high number, and this graph is zoomed!

4) **Pocket Perceptron**
The best hyperparameter in the case of Pocket Perceptron **Learning Rate = 1**.
The cross validation accuracy for this hyperparameter was: **0.89499**.
Over 20 epochs, the number of updates that are made to the training set total: **4480 updates**.

After those 20 epochs are over, the accuracy on the training set is: **0.91018**

And the accuracy on the test set is: **0.87626**!

Finally, if accuracy data is collected after every epoch, then the learning curve for pocket perceptron looks like this:



Note the accuracy starts at a high number, and this graph is zoomed!

Interestingly, averaged perceptron and pocket perceptron did the best job of "leveling off" out of all the different perceptron variants.

One more thing to note is, that some of the hyperparameters were very close to each other in averaged accuracy – so much so that, depending on the random seed, the 'best' hyperparameter could potentially change. In these cases, I'm not sure what the best way to tell what the best hyperparameter is, as there's no way to tell if the random seed just happened to be adversarial. To that end, I trained all my models on random seed 1 in Python, and simply judged based off of that, so at least all presented examples are consistent.