4 | The Perceptron

> Algebra is nothing more than geometry, in words; geometry is nothing more than algebra, in pictures. – Sophie Germain

Learning Objectives:

- Describe the biological motivation behind the perceptron.
- Classify learning algorithms based on whether they are error-driven or not.
- Implement the perceptron algorithm for binary classification.
- Draw perceptron weight vectors and the corresponding decision boundaries in two dimensions.
- Contrast the decision boundaries of decision trees, nearest neighbor algorithms and perceptrons.
- Compute the margin of a given weight vector on a given data set.

So far, you've seen two types of learning models: in decision trees, only a small number of features are used to make decisions; in nearest neighbor algorithms, all features are used equally. Neither of these extremes is always desirable. In some problems, we might want to use most of the features, but use some more than others.

In this chapter, we'll discuss the **perceptron** algorithm for learning **weights** for features. As we'll see, learning weights for features amounts to learning a **hyperplane** classifier: that is, basically a division of space into two halves by a straight line, where one half is "positive" and one half is "negative." In this sense, the perceptron can be seen as explicitly finding a good **linear decision boundary**.

Dependencies: Chapter 1, Chapter 3

4.1 *Bio-inspired Learning*

Folk biology tells us that our brains are made up of a bunch of little units, called **neurons**, that send electrical signals to one another. The *rate* of firing tells us how "activated" a neuron is. A single neuron, like that shown in Figure 4.1 might have three incoming neurons. These incoming neurons are firing at different rates (i.e., have different **activations**). Based on how much these incoming neurons are firing, and how "strong" the neural connections are, our main neuron will "decide" how strongly it wants to fire. And so on through the whole brain. Learning in the brain happens by neurons becoming connected to other neurons, and the strengths of connections adapting over time.

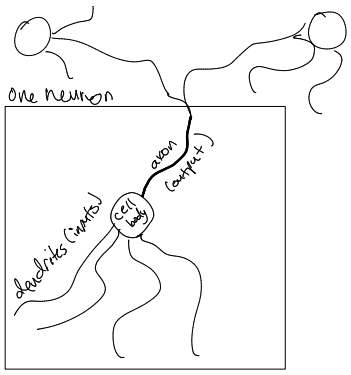


Figure 4.1: a picture of a neuron

The real biological world is much more complicated than this. However, our goal isn't to build a brain, but to simply be *inspired* by how they work. We are going to think of our learning algorithm as a *single* neuron. It receives input from D-many other neurons, one for each input feature. The strength of these inputs are the feature values. This is shown schematically in Figure 4.1. Each incoming connection has a weight and the neuron simply sums up all the weighted inputs. Based on this sum, it decides whether to "fire" or

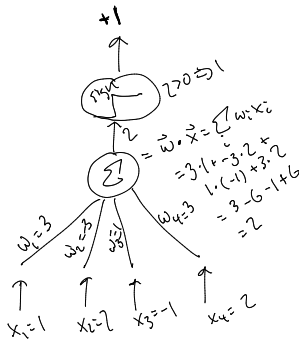


Figure 4.2: figure showing feature vector and weight vector and products and sum

not. Firing is interpreted as being a positive example and not firing is interpreted as being a negative example. In particular, if the weighted sum is positive, it "fires" and otherwise it doesn't fire. This is shown diagramatically in Figure 4.2.

Mathematically, an input vector $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_D \rangle$ arrives. The neuron stores D-many weights, $w_1, w_2, \ldots, w_D$. The neuron computes the sum:

$$a = \sum_{d=1}^{D} w_d x_d \tag{4.1}$$

to determine it's amount of "activation." If this activiation is positive (i.e., $a > 0$) it predicts that this example is a positive example. Otherwise it predicts a negative example.

The weights of this neuron are fairly easy to interpret. Suppose that a feature, for instance "is this a System's class?" gets a zero weight. Then the activation is the same regardless of the value of this feature. So features with zero weight are ignored. Features with positive weights are indicative of positive examples because they cause the activation to increase. Features with negative weights are indicative of negative examples because they cause the activiation to decrease.

It is often convenient to have a non-zero **threshold**. In other words, we might want to predict positive if $a > \theta$ for some value θ. The way that is most convenient to achieve this is to introduce a **bias** term into the neuron, so that the activation is always increased by some fixed value b. Thus, we compute:

$$a = \left[\sum_{d=1}^{D} w_d x_d \right] + b \tag{4.2}$$

? What would happen if we encoded binary features like "is this a System's class" as no=0 and yes=−1 (rather than the standard no=0 and yes=+1)?

? If you wanted the activation threshold to be $a > \theta$ instead of $a > 0$, what value would b have to be?

This is the complete neural model of learning. The model is parameterized by D-many weights, $w_1, w_2, \ldots, w_D$, and a single scalar bias value b.

4.2 *Error-Driven Updating: The Perceptron Algorithm*

The **perceptron** is a classic learning algorithm for the neural model of learning. Like K-nearest neighbors, it is one of those frustrating algorithms that is incredibly simple and yet works amazingly well, for some types of problems.

The algorithm is actually quite different than either the decision tree algorithm or the KNN algorithm. First, it is **online**. This means that instead of considering the *entire data set* at the same time, it only ever looks at one example. It processes that example and then goes

Algorithm 5 PERCEPTRONTRAIN($\mathbf{D}$, *MaxIter*)

```
1:  w_d ← 0, for all d = 1 ... D                    // initialize weights
2:  b ← 0                                           // initialize bias
3:  for iter = 1 ... MaxIter do
4:      for all (x,y) ∈ D do
5:          a ← Σ_{d=1}^{D} w_d x_d + b             // compute activation for this example
6:          if ya ≤ 0 then
7:              w_d ← w_d + y x_d, for all d = 1 ... D   // update weights
8:              b ← b + y                           // update bias
9:          end if
10:     end for
11: end for
12: return w_0, w_1, ..., w_D, b
```

Algorithm 6 PERCEPTRONTEST(w_0, w_1, ..., w_D, b, $\hat{\boldsymbol{x}}$)

```
1:  a ← Σ_{d=1}^{D} w_d x̂_d + b                     // compute activation for the test example
2:  return SIGN(a)
```

on to the next one. Second, it is **error driven**. This means that, so long as it is doing well, it doesn't bother updating its parameters.

The algorithm maintains a "guess" at good parameters (weights and bias) as it runs. It processes one example at a time. For a given example, it makes a prediction. It checks to see if this prediction is correct (recall that this is *training data*, so we have access to true labels). If the prediction is correct, it does nothing. Only when the prediction is incorrect does it change its parameters, and it changes them in such a way that it would do better on this example next time around. It then goes on to the next example. Once it hits the last example in the training set, it loops back around for a specified number of iterations.

The training algorithm for the perceptron is shown in Algorithm 4.2 and the corresponding prediction algorithm is shown in Algorithm 4.2. There is one "trick" in the training algorithm, which probably seems silly, but will be useful later. It is in line 6, when we check to see if we want to make an update or not. We want to make an update if the current prediction (just SIGN(a)) is incorrect. The trick is to multiply the true label y by the activation a and compare this against zero. Since the label y is either $+1$ or -1, you just need to realize that ya is positive whenever a and y have the same sign. In other words, the product ya is positive if the current prediction is correct.

The particular form of update for the perceptron is quite simple. The weight w_d is increased by yx_d and the bias is increased by y. The goal of the update is to adjust the parameters so that they are "better" for the current example. In other words, if we saw this example

? It is very very important to check $ya \leq 0$ rather than $ya < 0$. Why?

twice in a row, we should do a better job the second time around.

To see why this particular update achieves this, consider the following scenario. We have some current set of parameters $w_1, \ldots, w_D, b$. We observe an example $(\boldsymbol{x}, y)$. For simplicity, suppose this is a positive example, so $y = +1$. We compute an activation a, and make an error. Namely, $a < 0$. We now update our weights and bias. Let's call the new weights $w'_1, \ldots, w'_D, b'$. Suppose we observe the same example again and need to compute a new activation a'. We proceed by a little algebra:

$$a' = \sum_{d=1}^{D} w'_d x_d + b' \tag{4.3}$$

$$= \sum_{d=1}^{D} (w_d + x_d) x_d + (b + 1) \tag{4.4}$$

$$= \sum_{d=1}^{D} w_d x_d + b + \sum_{d=1}^{D} x_d x_d + 1 \tag{4.5}$$

$$= a + \sum_{d=1}^{D} x_d^2 + 1 \quad > \quad a \tag{4.6}$$

So the difference between the old activation a and the new activation a' is $\sum_d x_d^2 + 1$. But $x_d^2 \geq 0$, since it's squared. So this value is *always* at least one. Thus, the new activation is always at least the old activation plus one. Since this was a positive example, we have successfully moved the activation in the proper direction. (Though note that there's no guarantee that we will correctly classify this point the second, third or even fourth time around!)

? This analysis hold for the case positive examples ($y = +1$). It should also hold for negative examples. Work it out.

The only **hyperparameter** of the perceptron algorithm is *MaxIter*, the number of passes to make over the training data. If we make many many passes over the training data, then the algorithm is likely to overfit. (This would be like studying *too long* for an exam and just confusing yourself.) On the other hand, going over the data only one time might lead to underfitting. This is shown experimentally in Figure 4.3. The x-axis shows the number of passes over the data and the y-axis shows the training error and the test error. As you can see, there is a "sweet spot" at which test performance begins to degrade due to overfitting.

One aspect of the perceptron algorithm that is left underspecified is line 4, which says: loop over all the training examples. The natural implementation of this would be to loop over them in a constant order. The is actually a bad idea.

Consider what the perceptron algorithm would do on a data set that consisted of 500 positive examples followed by 500 negative examples. After seeing the first few positive examples (maybe five), it would likely decide that *every* example is positive, and would stop

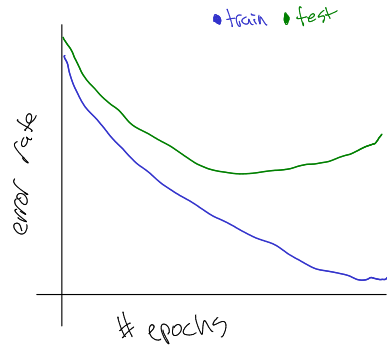


Figure 4.3: training and test error via early stopping

learning anything. It would do well for a while (next 495 examples), until it hit the batch of negative examples. Then it would take a while (maybe ten examples) before it would start predicting everything as negative. By the end of one pass through the data, it would really only have learned from a handful of examples (fifteen in this case).

So one thing you need to avoid is presenting the examples in some fixed order. This can easily be accomplished by permuting the order of examples once in the beginning and then cycling over the data set in the same (permuted) order each iteration. However, it turns out that you can actually do *better* if you re-permute the examples in each iteration. Figure 4.4 shows the effect of re-permuting on convergence speed. In practice, permuting each iteration tends to yield about 20% savings in number of iterations. In theory, you can actually prove that it's expected to be about twice as fast.

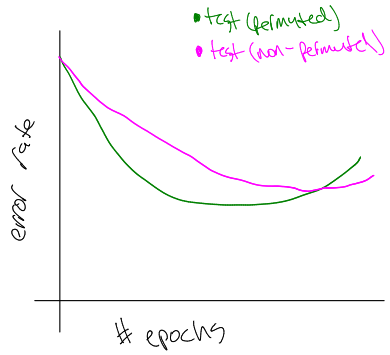


Figure 4.4: training and test error for permuting versus not-permuting

? If permuting the data each iteration saves somewhere between 20% and 50% of your time, are there any cases in which you might *not* want to permute the data every iteration?

4.3 Geometric Intrepretation

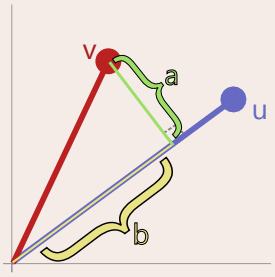
A question you should be asking yourself by now is: what does the decision boundary of a perceptron look like? You can actually answer that question mathematically. For a perceptron, the decision boundary is precisely where the sign of the activation, a, changes from -1 to $+1$. In other words, it is the set of points $\boldsymbol{x}$ that achieve *zero* activation. The points that are not clearly positive nor negative. For simplicity, we'll first consider the case where there is no "bias" term (or, equivalently, the bias is zero). Formally, the decision boundary $\mathcal{B}$ is:

$$\mathcal{B} = \left\{ \boldsymbol{x} : \sum_d w_d x_d = 0 \right\} \tag{4.7}$$

We can now apply some linear algebra. Recall that $\sum_d w_d x_d$ is just the **dot product** between the vector $\boldsymbol{w} = \langle w_1, w_2, \ldots, w_D \rangle$ and the vector $\boldsymbol{x}$. We will write this as $\boldsymbol{w} \cdot \boldsymbol{x}$. Two vectors have a zero dot product if and only if they are **perpendicular**. Thus, if we think of the weights as a vector $\boldsymbol{w}$, then the decision boundary is simply the plane perpendicular to $\boldsymbol{w}$.

MATH REVIEW | DOT PRODUCTS

Given two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ their dot product $\boldsymbol{u} \cdot \boldsymbol{v}$ is $\sum_d u_d v_d$. The dot product grows large and positive when $\boldsymbol{u}$ and $\boldsymbol{v}$ point in same direction, grows large and negative when $\boldsymbol{u}$ and $\boldsymbol{v}$ point in opposite directions, and is zero when their are perpendicular. A useful geometric interpretation of dot products is **projection**. Suppose $||\boldsymbol{u}|| = 1$, so that $\boldsymbol{u}$ is a **unit vector**. We can think of any other vector $\boldsymbol{v}$ as consisting of two components: (a) a component in the direction of $\boldsymbol{u}$ and (b) a component that's perpendicular to $\boldsymbol{u}$. This is depicted geometrically to the right: Here, $\boldsymbol{u} = \langle 0.8, 0.6 \rangle$ and $\boldsymbol{v} = \langle 0.37, 0.73 \rangle$. We can think of $\boldsymbol{v}$ as the sum of two vectors, $\boldsymbol{a}$ and $\boldsymbol{b}$, where $\boldsymbol{a}$ is parallel to $\boldsymbol{u}$ and $\boldsymbol{b}$ is perpendicular. The length of $\boldsymbol{b}$ is exactly $\boldsymbol{u} \cdot \boldsymbol{v} = 0.734$, which is why you can think of dot products as projections: the dot product between $\boldsymbol{u}$ and $\boldsymbol{v}$ is the "projection of $\boldsymbol{v}$ onto $\boldsymbol{u}$."



This is shown pictorially in Figure 4.6. Here, the weight vector is shown, together with it's perpendicular plane. This plane forms the decision boundary between positive points and negative points. The vector points in the direction of the positive examples and away from the negative examples.

One thing to notice is that the *scale* of the weight vector is irrelevant from the perspective of classification. Suppose you take a weight vector $\boldsymbol{w}$ and replace it with $2\boldsymbol{w}$. All activations are now doubled. But their sign does not change. This makes complete sense geometrically, since all that matters is which side of the plane a test point falls on, now how far it is from that plane. For this reason, it is common to work with **normalized** weight vectors, $\boldsymbol{w}$, that have length one; i.e., $||\boldsymbol{w}|| = 1$.

The geometric intuition can help us even more when we realize that dot products compute projections. That is, the value $\boldsymbol{w} \cdot \boldsymbol{x}$ is just the distance of $\boldsymbol{x}$ from the origin when projected *onto* the vector $\boldsymbol{w}$. This is shown in Figure 4.7. In that figure, all the data points are projected onto $\boldsymbol{w}$. Below, we can think of this as a one-dimensional version of the data, where each data point is placed according to its projection along $\boldsymbol{w}$. This distance along $\boldsymbol{w}$ is exactly the *activiation* of that example, with no bias.

From here, you can start thinking about the role of the bias term. Previously, the threshold would be at zero. Any example with a negative projection onto $\boldsymbol{w}$ would be classified negative; any example with a positive projection, positive. The bias simply moves this threshold. Now, after the projection is computed, b is added to get the overall activation. The projection *plus* b is then compared against

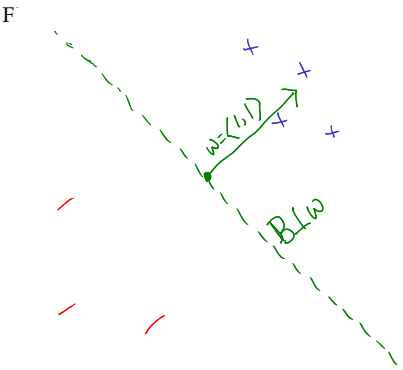


Figure 4.6: picture of data points with hyperplane and weight vector

? If I give you a non-zero weight vector $\boldsymbol{w}$, how do I compute a weight vector $\boldsymbol{w}'$ that points in the same direction but has a norm of one?

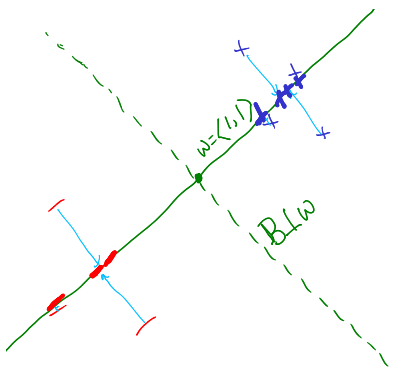


Figure 4.7: The same picture as before, but with projections onto weight vector; then, below, those points along a one-dimensional axis with zero marked.

zero.

Thus, from a geometric perspective, the role of the bias is to *shift* the decision boundary away from the origin, in the direction of $\boldsymbol{w}$. It is shifted exactly $-b$ units. So if b is positive, the boundary is shifted away from $\boldsymbol{w}$ and if b is negative, the boundary is shifted toward $\boldsymbol{w}$. This is shown in Figure 4.8. This makes intuitive sense: a positive bias means that more examples should be classified positive. By moving the decision boundary in the negative direction, more space yields a positive classification.

The decision boundary for a perceptron is a very magical thing. In D dimensional space, it is always a $D-1$-dimensional hyperplane. (In two dimensions, a 1-d hyperplane is simply a line. In three dimensions, a 2-d hyperplane is like a sheet of paper.) This hyperplane divides space in half. In the rest of this book, we'll refer to the weight vector, and to hyperplane it defines, interchangeably.

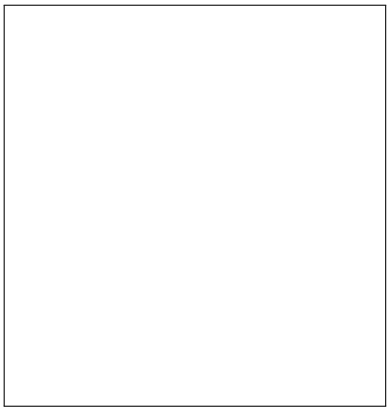


Figure 4.8: perc:bias: perceptron picture with bias

The perceptron update can also be considered geometrically. (For simplicity, we will consider the **unbiased** case.) Consider the situation in Figure 4.9. Here, we have a current guess as to the hyperplane, and positive training example comes in that is currently mis-classified. The weights are updated: $\boldsymbol{w} \leftarrow \boldsymbol{w} + y\boldsymbol{x}$. This yields the new weight vector, also shown in the Figure. In this case, the weight vector changed enough that this training example is now correctly classified.

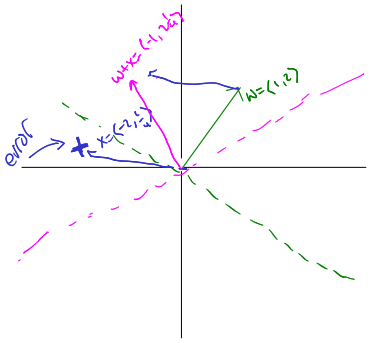


Figure 4.9: perceptron picture with update, no bias

4.4 *Interpreting Perceptron Weights*

You may find yourself having run the perceptron, learned a really awesome classifier, and then wondering "what the heck is this classifier doing?" You might ask this question because you're curious to learn something about the underlying data. You might ask this question because you want to make sure that the perceptron is learning "the right thing." You might ask this question because you want to remove a bunch of features that aren't very useful because they're expensive to compute or take a lot of storage.

The perceptron learns a classifier of the form $\boldsymbol{x} \mapsto \text{sign}\left(\sum_d w_d x_d + b\right)$. A reasonable question to ask is: how sensitive is the final classification to *small changes* in some particular feature. We can answer this question by taking a derivative. If we arbitrarily take the 7th feature we can compute $\frac{\partial}{\partial x_7}\left(\sum_d w_d x_d + b\right) = w_7$. This says: the rate at which the activation changes as a function of the 7th feature is exactly w_7. This gives rise to a useful heuristic for interpreting perceptron weights: **sort all the weights from largest (positive) to largest (negative), and take the top ten and bottom ten**. The top ten are the features that the perceptron is most sensitive to for making positive

predictions. The bottom ten are the features that the perceptron is most sensitive to for making negative predictions.

This heuristic is useful, especially when the inputs $\boldsymbol{x}$ consist entirely of binary values (like a bag of words representation). The heuristic is less useful when the range of the individual features varies significantly. The issue is that if you have one feat x_5 that's either 0 or 1, and another feature x_7 that's either 0 or 100, but $w_5 = w_7$, it's reasonable to say that w_7 is more important because it is likely to have a much larger influence on the final prediction. The easiest way to compensate for this is simply to scale your features ahead of time: this is another reason why feature scaling is a useful preprocessing step.

4.5 *Perceptron Convergence and Linear Separability*

You already have an intuitive feeling for why the perceptron works: it moves the decision boundary in the direction of the training examples. A question you should be asking yourself is: does the perceptron converge? If so, what does it converge to? And how long does it take?

It is easy to construct data sets on which the perceptron algorithm will never converge. In fact, consider the (very uninteresting) learning problem with *no features*. You have a data set consisting of one positive example and one negative example. Since there are no features, the only thing the perceptron algorithm will ever do is adjust the bias. Given this data, you can run the perceptron for a bajillion iterations and it will never settle down. As long as the bias is non-negative, the negative example will cause it to decrease. As long as it is non-positive, the positive example will cause it to increase. Ad infinitum. (Yes, this is a very contrived example.)

What does it mean for the perceptron to converge? It means that it can make an entire pass through the training data without making *any* more updates. In other words, it has correctly classified *every* training example. Geometrically, this means that it was found some hyperplane that correctly segregates the data into positive and negative examples, like that shown in Figure 4.10.

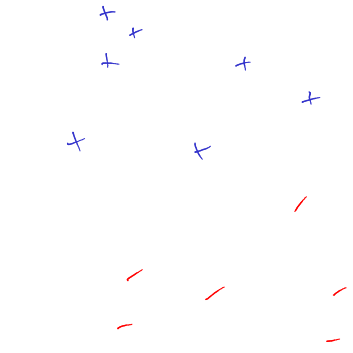


Figure 4.10: separable data

In this case, this data is **linearly separable**. This means that there exists *some* hyperplane that puts all the positive examples on one side and all the negative examples on the other side. If the training is *not* linearly separable, like that shown in Figure 4.11, then the perceptron has no hope of converging. It could never possibly classify each point correctly.

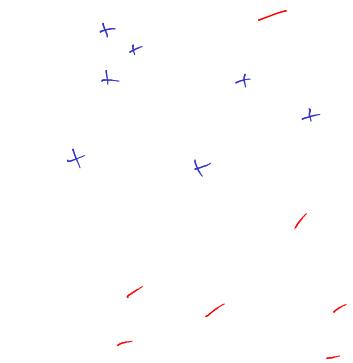


Figure 4.11: inseparable data

The somewhat surprising thing about the perceptron algorithm is that *if* the data is linearly separable, *then* it will converge to a weight

vector that separates the data. (And if the data is inseparable, then it will never converge.) This is great news. It means that the perceptron converges whenever it is even remotely possible to converge.

The second question is: how long does it take to converge? By "how long," what we really mean is "how many updates?" As is the case for much learning theory, you will not be able to get an answer of the form "it will converge after 5293 updates." This is asking too much. The sort of answer we can hope to get is of the form "it will converge after *at most* 5293 updates."

What you might expect to see is that the perceptron will converge more quickly for easy learning problems than for hard learning problems. This certainly fits intuition. The question is how to *define* "easy" and "hard" in a meaningful way. One way to make this definition is through the notion of **margin**. If I give you a data set and hyperplane that separates itthen the *margin* is the distance between the hyperplane and the nearest point. Intuitively, problems with large margins should be easy (there's lots of "wiggle room" to find a separating hyperplane); and problems with small margins should be hard (you really have to get a very specific well tuned weight vector).

Formally, given a data set $\mathbf{D}$, a weight vector $\boldsymbol{w}$ and bias b, the margin of $\boldsymbol{w}, b$ on $\mathbf{D}$ is defined as:

$$\mathit{margin}(\mathbf{D}, \boldsymbol{w}, b) = \begin{cases} \min_{(\boldsymbol{x},y)\in\mathbf{D}} y(\boldsymbol{w}\cdot\boldsymbol{x}+b) & \text{if } \boldsymbol{w} \text{ separates } \mathbf{D} \\ -\infty & \text{otherwise} \end{cases} \tag{4.8}$$

In words, the margin is only defined if $\boldsymbol{w}, b$ actually separate the data (otherwise it is just $-\infty$). In the case that it separates the data, we find the point with the minimum activation, after the activation is multiplied by the label.

? So long as the margin is not $-\infty$, it is always positive. Geometrically this makes sense, but why does Eq (4.8) yield this?

For some historical reason (that is unknown to the author), margins are always denoted by the Greek letter γ (gamma). One often talks about the **margin of a data set**. The margin of a data set is the largest attainable margin on this data. Formally:

$$\mathit{margin}(\mathbf{D}) = \sup_{\boldsymbol{w},b} \mathit{margin}(\mathbf{D}, \boldsymbol{w}, b) \tag{4.9}$$

In words, to compute the margin of a data set, you "try" every possible $\boldsymbol{w}, b$ pair. For each pair, you compute its margin. We then take the largest of these as the overall margin of the data.[1] If the data is not linearly separable, then the value of the sup, and therefore the value of the margin, is $-\infty$.

There is a famous theorem due to Rosenblatt[2] that shows that the number of errors that the perceptron algorithm makes is bounded by γ^{-2}. More formally:

[1] You can read "sup" as "max" if you like: the only difference is a technical difference in how the $-\infty$ case is handled.

[2] Rosenblatt 1958

Theorem 2 (Perceptron Convergence Theorem). *Suppose the perceptron algorithm is run on a linearly separable data set* $\mathbf{D}$ *with margin* $\gamma > 0$. *Assume that* $||\boldsymbol{x}|| \leq 1$ *for all* $\boldsymbol{x} \in \mathbf{D}$. *Then the algorithm will converge after at most* $\frac{1}{\gamma^2}$ *updates.*

The proof of this theorem is elementary, in the sense that it does not use any fancy tricks: it's all just algebra. The *idea* behind the proof is as follows. If the data is linearly separable with margin γ, then there exists some weight vector $\boldsymbol{w}^*$ that achieves this margin. Obviously we don't know what $\boldsymbol{w}^*$ is, but we know it exists. The perceptron algorithm is trying to find a weight vector $\boldsymbol{w}$ that points roughly in the same direction as $\boldsymbol{w}^*$. (For large γ, "roughly" can be very rough. For small γ, "roughly" is quite precise.) Every time the perceptron makes an update, the angle between $\boldsymbol{w}$ and $\boldsymbol{w}^*$ changes. What we prove is that the angle actually *decreases.* We show this in two steps. First, the dot product $\boldsymbol{w} \cdot \boldsymbol{w}^*$ increases a lot. Second, the norm $||\boldsymbol{w}||$ does not increase very much. Since the dot product is increasing, but $\boldsymbol{w}$ isn't getting too long, the angle between them has to be shrinking. The rest is algebra.

Proof of Theorem 2. The margin $\gamma > 0$ must be realized by some set of parameters, say $\boldsymbol{x}^*$. Suppose we train a perceptron on this data. Denote by $\boldsymbol{w}^{(0)}$ the initial weight vector, $\boldsymbol{w}^{(1)}$ the weight vector after the *first update*, and $\boldsymbol{w}^{(k)}$ the weight vector after the kth update. (We are essentially ignoring data points on which the perceptron doesn't update itself.) First, we will show that $\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)}$ grows quicky as a function of k. Second, we will show that $\left|\left|\boldsymbol{w}^{(k)}\right|\right|$ does not grow quickly.

First, suppose that the kth update happens on example $(\boldsymbol{x}, y)$. We are trying to show that $\boldsymbol{w}^{(k)}$ is becoming aligned with $\boldsymbol{w}^*$. Because we updated, know that this example was misclassified: $y\boldsymbol{w}^{(k-1)} \cdot \boldsymbol{x} < 0$. After the update, we get $\boldsymbol{w}^{(k)} = \boldsymbol{w}^{(k-1)} + y\boldsymbol{x}$. We do a little computation:

$$\begin{aligned}
\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)} &= \boldsymbol{w}^* \cdot \left(\boldsymbol{w}^{(k-1)} + y\boldsymbol{x}\right) && \text{definition of } \boldsymbol{w}^{(k)} && (4.10)\\
&= \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k-1)} + y\boldsymbol{w}^* \cdot \boldsymbol{x} && \text{vector algebra} && (4.11)\\
&\geq \boldsymbol{w}^* \cdot \boldsymbol{w}^{(k-1)} + \gamma && \boldsymbol{w}^* \text{ has margin } \gamma && (4.12)
\end{aligned}$$

Thus, every time $\boldsymbol{w}^{(k)}$ is updated, its projection onto $\boldsymbol{w}^*$ increases by at least γ. Therefore: $\boldsymbol{w}^* \cdot \boldsymbol{w}^{(k)} \geq k\gamma$.

Next, we need to show that the increase of γ along $\boldsymbol{w}^*$ occurs because $\boldsymbol{w}^{(k)}$ is getting closer to $\boldsymbol{w}^*$, not just because it's getting exceptionally long. To do this, we compute the norm of $\boldsymbol{w}^{(k)}$:

$$\left|\left|\boldsymbol{w}^{(k)}\right|\right|^2$$

$$= \left|\left| w^{(k-1)} + yx \right|\right|^2 \qquad \text{def. of } w^{(k)} \quad (4.13)$$

$$= \left|\left| w^{(k-1)} \right|\right|^2 + y^2 ||x||^2 + 2y w^{(k-1)} \cdot x \qquad \text{quadratic rule} \quad (4.14)$$

$$\leq \left|\left| w^{(k-1)} \right|\right|^2 + 1 + 0 \qquad \text{assumption and } a < 0 \quad (4.15)$$

Thus, the squared norm of $w^{(k)}$ increases by at most one every update. Therefore: $||w^{(k)}||^2 \leq k$.

Now we put together the two things we have learned before. By our first conclusion, we know $w^* \cdot w^{(k)} \geq k\gamma$. But our second conclusion, $\sqrt{k} \geq ||w^{(k)}||^2$. Finally, because w^* is a unit vector, we know that $||w^{(k)}|| \geq w^* \cdot w^{(k)}$. Putting this together, we have:

$$\sqrt{k} \quad \geq \quad \left|\left| w^{(k)} \right|\right| \quad \geq \quad w^* \cdot w^{(k)} \quad \geq \quad k\gamma \qquad (4.16)$$

Taking the left-most and right-most terms, we get that $\sqrt{k} \geq k\gamma$. Dividing both sides by k, we get $\frac{1}{\sqrt{k}} \geq \gamma$ and therefore $k \leq \frac{1}{\gamma^2}$. This means that once we've made $\frac{1}{\gamma^2}$ updates, we cannot make any more! □

It is important to keep in mind what this proof shows and what it does not show. It shows that if I give the perceptron data that is linearly separable with margin $\gamma > 0$, then the perceptron will converge to a solution that separates the data. And it will converge quickly when γ is large. It does not say anything about the solution, *other than* the fact that it separates the data. In particular, the proof makes use of the maximum margin separator. But the perceptron is not guaranteed to *find* this maximum margin separator. The data may be separable with margin 0.9 and the perceptron might still find a separating hyperplane with a margin of only 0.000001. Later (in Chapter 8), we will see algorithms that explicitly try to find the maximum margin solution.

? Perhaps we don't want to assume that all x have norm at most 1. If they have all have norm at most R, you can achieve a very similar bound. Modify the perceptron convergence proof to handle this case.

? Why does the perceptron convergence bound not contradict the earlier claim that poorly ordered data points (e.g., all positives followed by all negatives) will cause the perceptron to take an astronomically long time to learn?

4.6 *Improved Generalization: Voting and Averaging*

In the beginning of this chapter, there was a comment that the perceptron works amazingly well. This was a half-truth. The "vanilla" perceptron algorithm does well, but not *amazingly* well. In order to make it more competitive with other learning algorithms, you need to modify it a bit to get better generalization. The key issue with the vanilla perceptron is that *it counts later points more than it counts earlier points*.

To see why, consider a data set with 10,000 examples. Suppose that after the first 100 examples, the perceptron has learned a really

good classifier. It's so good that it goes over the next 9899 examples without making *any* updates. It reaches the 10,000th example and makes an error. It updates. For all we know, the update on this 10,000th example *completely ruins* the weight vector that has done so well on 99.99% of the data!

What we would like is for weight vectors that "survive" a long time to get more say than weight vectors that are overthrown quickly. One way to achieve this is by **voting**. As the perceptron learns, it remembers how long each hyperplane survives. At test time, each hyperplane encountered during training "votes" on the class of a test example. If a particular hyperplane survived for 20 examples, then it gets a vote of 20. If it only survived for one example, it only gets a vote of 1. In particular, let $(\boldsymbol{w}, b)^{(1)}, \ldots, (\boldsymbol{w}, b)^{(K)}$ be the $K+1$ weight vectors encountered during training, and $c^{(1)}, \ldots, c^{(K)}$ be the survival times for each of these weight vectors. (A weight vector that gets immediately updated gets $c = 1$; one that survives another round gets $c = 2$ and so on.) Then the prediction on a test point is:

$$\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c^{(k)} \text{sign}\left(\boldsymbol{w}^{(k)} \cdot \hat{\boldsymbol{x}} + b^{(k)}\right)\right) \tag{4.17}$$

This algorithm, known as the **voted perceptron** works quite well in practice, and there is some nice theory showing that it is guaranteed to generalize better than the vanilla perceptron. Unfortunately, it is also completely impractical. If there are 1000 updates made during perceptron learning, the voted perceptron requires that you store 1000 weight vectors, together with their counts. This requires an absurd amount of storage, and makes prediction 1000 times slower than the vanilla perceptron.

A much more practical alternative is the **averaged perceptron**. The idea is similar: you maintain a collection of weight vectors and survival times. However, at test time, you predict according to the *average* weight vector, rather than the voting. In particular, the prediction is:

$$\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c^{(k)} \left(\boldsymbol{w}^{(k)} \cdot \hat{\boldsymbol{x}} + b^{(k)}\right)\right) \tag{4.18}$$

The only difference between the voted prediction, Eq (4.17), and the averaged prediction, Eq (4.18), is the presense of the interior sign operator. With a little bit of algebra, we can rewrite the test-time prediction as:

$$\hat{y} = \text{sign}\left(\left(\sum_{k=1}^{K} c^{(k)} \boldsymbol{w}^{(k)}\right) \cdot \hat{\boldsymbol{x}} + \sum_{k=1}^{K} c^{(k)} b^{(k)}\right) \tag{4.19}$$

The advantage of the averaged perceptron is that we can simply maintain a *running sum* of the averaged weight vector (the blue term)

? The *training algorithm* for the voted perceptron is the same as the vanilla perceptron. In particular, in line 5 of Algorithm 4.2, the activation on a training example is computed based on the *current weight vector*, not based on the voted prediction. Why?

Algorithm 7 AVERAGEDPERCEPTRONTRAIN(**D**, *MaxIter*)

```
w ← ⟨0, 0, ... 0⟩   ,   b ← 0                    // initialize weights and bias
u ← ⟨0, 0, ... 0⟩   ,   β ← 0                    // initialize cached weights and bias
c ← 1                                            // initialize example counter to one
for iter = 1 ... MaxIter do
    for all (x,y) ∈ D do
        if y(w · x + b) ≤ 0 then
            w ← w + y x                          // update weights
            b ← b + y                            // update bias
            u ← u + y c x                        // update cached weights
            β ← β + y c                          // update cached bias
        end if
        c ← c + 1                                // increment counter regardless of update
    end for
end for
return w - (1/c) u, b - (1/c) β                  // return averaged weights and bias
```

and averaged bias (the red term). Test-time prediction is then just as efficient as it is with the vanilla perceptron.

The full training algorithm for the averaged perceptron is shown in Algorithm 4.6. Some of the notation is changed from the original perceptron: namely, vector operations are written as vector operations, and the activation computation is folded into the error checking.

It is probably not immediately apparent from Algorithm 4.6 that the computation unfolding is precisely the calculation of the averaged weights and bias. The most *natural* implementation would be to keep track of an averaged weight vector $\boldsymbol{u}$. At the end of every example, you would increase $\boldsymbol{u} \leftarrow \boldsymbol{u} + \boldsymbol{w}$ (and similarly for the bias). However, such an implementation would require that you updated the averaged vector on *every* example, rather than just on the examples that were incorrectly classified! Since we hope that eventually the perceptron learns to do a good job, we would hope that it will not make updates on every example. So, ideally, you would like to only update the averaged weight vector when the actual weight vector changes. The slightly clever computation in Algorithm 4.6 achieves this.

The averaged perceptron is almost always better than the perceptron, in the sense that it generalizes better to test data. However, that does not free you from having to do **early stopping**. It will, eventually, overfit.

? By writing out the computation of the averaged weights from Eq (4.18) as a telescoping sum, derive the computation from Algorithm 4.6.

4.7 *Limitations of the Perceptron*

Although the perceptron is very useful, it is fundamentally limited in a way that neither decision trees nor KNN are. Its limitation is that

its decision boundaries can *only* be linear. The classic way of showing this limitation is through the XOR problem (XOR = exclusive or). The XOR problem is shown graphically in Figure 4.12. It consists of four data points, each at a corner of the unit square. The labels for these points are the same, along the diagonals. You can try, but you will not be able to find a linear decision boundary that perfectly separates these data points.

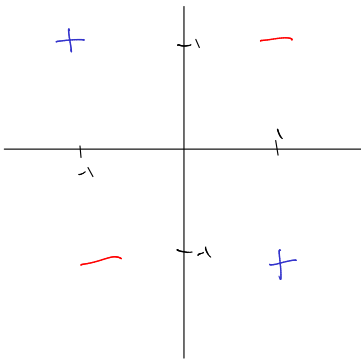


Figure 4.12: picture of xor problem

One question you might ask is: do XOR-like problems exist in the real world? Unfortunately for the perceptron, the answer is yes. Consider a sentiment classification problem that has three features that simply say whether a given word is contained in a review of a course. These features are: EXCELLENT, TERRIBLE and NOT. The EXCELLENT feature is indicative of positive reviews and the TERRIBLE feature is indicative of negative reviews. But in the presence of the NOT feature, this categorization flips.

One way to address this problem is by adding **feature combinations**. We could add two additional features: EXCELLENT-AND-NOT and TERRIBLE-AND-NOT that indicate a conjunction of these base features. By assigning weights as follows, you can achieve the desired effect:

$$w_{\text{EXECELLENT}} = +1 \qquad w_{\text{TERRIBLE}} = -1 \qquad w_{\text{NOT}} = 0$$
$$w_{\text{EXECLLENT-AND-NOT}} = -2 \qquad w_{\text{TERRIBLE-AND-NOT}} = +2$$

In this particular case, we have addressed the problem. However, if we start with D-many features, if we want to add all pairs, we'll blow up to $\binom{D}{2} = \mathcal{O}(D^2)$ features through this **feature mapping**. And there's no guarantee that pairs of features is enough. We might need triples of features, and now we're up to $\binom{D}{3} = \mathcal{O}(D^2)$ features. These additional features will drastically increase computation and will often result in a stronger propensity to overfitting.

? Suppose that you took the XOR problem and added one new feature: $x_3 = x_1 \wedge x_2$ (the logical and of the two existing features). Write out feature weights and a bias that would achieve perfect classification on this data.

In fact, the "XOR problem" is so significant that it basically killed research in classifiers with linear decision boundaries for a decade or two. Later in this book, we will see two alternative approaches to taking key ideas from the perceptron and generating classifiers with non-linear decision boundaries. One approach is to combine multiple perceptrons in a single framework: this is the **neural networks** approach (see Chapter 10). The second approach is to find computationally efficient ways of doing feature mapping in a computationally and statistically efficient way: this is the **kernels** approach (see Chapter 11).

4.8 Further Reading

TODO further reading