Linear Classifiers and the Perceptron

William Cohen

February 4, 2008

1 Linear classifiers

Let's assume that every *instance* is an n-dimensional vector of real numbers $\mathbf{x} \in \mathcal{R}^n$, and there are only two possible classes, $y = (+1)$ and $y = (-1)$, so every *example* is a pair $(\mathbf{x}, y)$. (Notation: I will use **boldface** to indicate vectors here, so $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$). A *linear classifier* is a vector $\mathbf{w}$ that makes the prediction

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} w_i x_i)$$

where $\text{sign}(x) = +1$ if $x \geq 0$ and $\text{sign}(x) = -1$ if $x < 0$. [1] If you remember your linear algebra this weighted sum of x_i's is called the *inner product* of $\mathbf{w}$ and $\mathbf{x}$ and it's usually written $\mathbf{w} \cdot \mathbf{x}$, so this classifier can be written even more compactly as

$$\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$$

Visually, for a vector $\mathbf{w}$, $\mathbf{x} \cdot \mathbf{w}$ is the distance of the result you get "if you project $\mathbf{x}$ onto $\mathbf{w}$" (see Figure 1).

It might seem that representing examples as real-number vectors is somewhat constraining. It seems fine if your attributes are numeric (e.g., "Temperature=72") but what if you have an attribute "Outlook" with three possible discrete values "Rainy", "Sunny", and "Cloudy"? One answer is to replace this single attribute with three binary attributes: one that set to 1 when the outlook is rainy, and zero otherwise; one that set to 1 when the outlook is sunny, and zero otherwise; and one that set to 1 when the outlook is cloudy, and zero otherwise. So a dataset like the one below would be converted to examples in $\mathcal{R}^4$ as shown:

	Outlook	Temp	PlayTennis?		
Day1	Rainy	85	No	$\longrightarrow$	$(\langle 1, 0, 0, 85 \rangle, -1)$
Day2	Sunny	87	No	$\longrightarrow$	$(\langle 1, 0, 0, 87 \rangle, -1)$
Day3	Cloudy	75	Yes	$\longrightarrow$	$(\langle 0, 0, 1, 75 \rangle, +1)$

[1] This is a little different from the usual definition, where $\text{sign}(0) = 0$, but I'd rather not have to deal with the question of what to predict when $\sum_{i=1}^{n} w_i x_i = 0$.

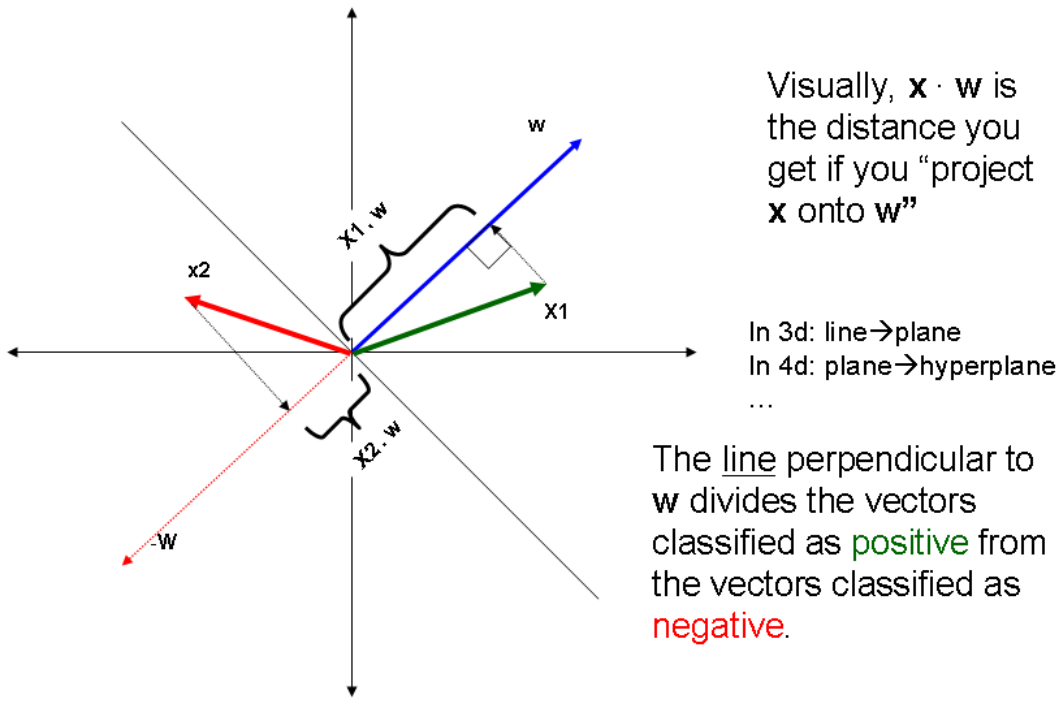

Figure 1: A geometric view of the inner product.

Another answer, of course, is to abandon these discrete values and instead focus on numeric attributes—e.g., let "Outlook" be encoded as a real number representing the probability of rain, so that Day2 would be encoded as $\langle 0.01, 87 \rangle$, where the 0.01 means a 1/100 chance of rain.

However, encoding your examples as pure numeric vectors vectors has one small problem. As I've defined it, a linear classifier is doomed to predict $\hat{y} = 1$ on a perfectly sunny day (Outlook=0) if the temperature is also zero—regardless of what weight vector $\mathbf{w}$ you pick, $\mathbf{w} \cdot \langle 0, 0 \rangle$ will be zero, and $\hat{y}$ will be one. Since zero-degree weather isn't conducive to playing tennis, no matter how clear it is, we can add one more trick to our encoding of examples and add an extra dimension to each vector, with a value that is *always* one. Let's label that extra dimension 0: then

$$\hat{y} = \text{sign}(w_0 x_0 + \sum_{i=1}^{n} w_i x_i) = \text{sign}(w_0 + \sum_{i=1}^{n} w_i x_i) \tag{1}$$

the second half of the equation holding because for every example $\mathbf{x}$, $x_0 = 1$.

This trick gives our linear classifier a bit more expressive power, and we can still write our classifier using the super-compact notation $\hat{y} = \text{sign}(\mathbf{wx})$ if we like. The weight w_0 is sometimes called a *bias term*.

2 Naive Bayes is a linear classifer

How do you learn a linear classifier? Well, you already know one way. To make things simple, I'll assume that $\mathbf{x}$ is not just a real-valued vector, but is a *binary* vector, in the discussion below.

You remember that Naive Bayes can be written as follows:

$$\hat{y} = \text{argmax}_y P(y|\mathbf{x}) = \text{argmax}_y P(\mathbf{x}|y)P(y) = \text{argmax}_y \prod_{i=1}^{n} P(x_i|y)P(y)$$

Since the log function is monotonic we can write this as

$$\hat{y} = \text{argmax}_y \log \prod_{i=1}^{n} P(x_i|y)P(y) = \text{argmax}_y \left(\sum_{i=1}^{n} \log P(x_i|y) + \log P(y) \right)$$

And if there are only two classes, $y = +1$ and $y = -1$, we can write this as

$$\hat{y} = \text{sign} \left[\left(\sum_{i=1}^{n} \log P(x_i|\, Y{=}{+}1) + \log P(\, Y{=}{+}1) \right) - \left(\sum_{i=1}^{n} \log P(x_i|\, Y{=}\text{-}1) + \log P(\, Y{=}\text{-}1) \right) \right]$$

which we can rearrange as

$$\hat{y} = \text{sign} \left(\sum_{i=1}^{n} (\log P(x_i|\, Y{=}{+}1) - \log P(x_i|\, Y{=}\text{-}1)) + (\log P(\, Y{=}{+}1) - \log P(\, Y{=}\text{-}1)) \right)$$

and if we use the fact that $\log x - \log(y) = \log \frac{x}{y}$, we can write this as[2]

$$\hat{y} = \text{sign} \left(\sum_{i=1}^{n} \log \frac{P(x_i|\, Y{=}{+}1)}{P(x_i|\, Y{=}\text{-}1)} + \log \frac{P(\, Y{=}{+}1)}{P(\, Y{=}\text{-}1)} \right) \tag{2}$$

This is starting to look a little more linear! To finish the job, let's think about what this means. When we say $\log \frac{P(x_i|\, Y{=}{+}1)}{P(x_i|\, Y{=}\text{-}1)}$, we're using that to describe a *function of* x_i, which could be written out as

$$\log \frac{P(x_i|\, Y{=}{+}1)}{P(x_i|\, Y{=}\text{-}1)} \equiv f(x_i) \equiv \begin{cases} \log \frac{P(X_i=1|\, Y{=}{+}1)}{P(X_i=1|\, Y{=}\text{-}1)} & \text{if } x_i = 1 \\ \log \frac{P(X_i=0|\, Y{=}{+}1)}{P(X_i=0|\, Y{=}\text{-}1)} & \text{if } x_i = 0 \end{cases} \tag{3}$$

To make the next few equations uncluttered let's define p_i and q_i as

$$p_i \equiv \log \frac{P(X_i = 1|\, Y{=}{+}1)}{P(X_i = 1|\, Y{=}\text{-}1)}$$

$$q_i \equiv \log \frac{P(X_i = 0|\, Y{=}{+}1)}{P(X_i = 0|\, Y{=}\text{-}1)}$$

[2]As an aside, expressions like $o = \log \frac{P(\, Y{=}{+}1)}{P(\, Y{=}\text{-}1)}$ are called *log odds*, and they mean something. If the logs are base 2 and $o = 3$, then the event $Y{=}{+}1$ is $2^3 = 8$ times as likely as the event $Y{=}\text{-}1$, while if $o = -4$ then the event $Y{=}\text{-}1$ is about $2^4 = 16$ times as likely as the event $Y{=}{+}1$.

A slightly tricky way to get rid of the if-thens in Equation 3 is to write it as

$$f(x_i) \equiv x_i p_i + (1 - x_i) q_i$$

(This is essentially the same trick as Tom used in deriving the MLE for a binomial - do you see why?) This of course can be written as

$$f(x_i) = x_i(p_i - q_i) + q_i \tag{4}$$

and then plugging Equation 4 into Equation 2 we get

$$\hat{y} = \text{sign}\left(\sum_{i=1}^{n}(x_i(p_i - q_i) + q_i) + \log \frac{P(\,Y{=}{+}1)}{P(\,Y{=}\text{-}1)}\right) \tag{5}$$

Now, we're almost done. Let's define $w_i = p_i - q_i$ and define :

$$\begin{aligned} w_i &\equiv p_i - q_i \\ w_0 &\equiv \sum_i q_i + \log \frac{P(\,Y{=}{+}1)}{P(\,Y{=}\text{-}1)} \end{aligned}$$

where w_0 is that "bias term" we used in Equation 1. Now the Naive Bayes prediction from Equation 5 becomes

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} x_i w_i + w_0)$$

Putting it all together: for binary vectors $\mathbf{x}' = \langle x_1, \ldots, x_n \rangle$ Naive Bayes can be implemented as a linear classifier, to be applied to the augmented example $\mathbf{x} = \langle x_0 = 1, x_1, \ldots, x_n \rangle$. The weight vector $\mathbf{w}$ has this form:

$$\begin{aligned} w_i &\equiv \log \frac{P(X_i = 1|\,Y{=}{+}1)}{P(X_i = 1|\,Y{=}\text{-}1)} - \log \frac{P(X_i = 0|\,Y{=}{+}1)}{P(X_i = 0|\,Y{=}\text{-}1)} \\ w_0 &\equiv \sum_i (\log \frac{P(X_i = 0|\,Y{=}{+}1)}{P(X_i = 0|\,Y{=}\text{-}1)}) + \log \frac{P(\,Y{=}{+}1)}{P(\,Y{=}\text{-}1)} \end{aligned}$$

and the Naive Bayes classification is

$$\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$$

3 Online learning for classification

3.1 Online learning

Bayesian probability is the most-studied mathematical model of learning. But there are other models that also are experimentally successful, and give useful insight. Sometimes these models are also mathematically more appropriate: e.g., even if all the probabilistic assumptions are correct, a MAP estimate might not minimize prediction errors.

Another useful model is on-line learning. In this document, I'll consider *on-line learners* that can do two things. The first is to make a *prediction* $\hat{y}$ on an example $\mathbf{x}$, where $\hat{y} \in \{-1, +1\}$. The second is to *update* the learner's state, by "accepting" a new example $\langle \mathbf{x}, y \rangle$.

As some background, remember that if θ is the angle between $\mathbf{x}$ and $\mathbf{u}$,

$$\cos\theta = \frac{\mathbf{x} \cdot \mathbf{u}}{\|\mathbf{x}\|\|\mathbf{u}\|}$$

This is important if you're trying to keep a picture in your mind of what all these dot-products mean. A special case is that $\mathbf{x} \cdot \mathbf{u} = \|\mathbf{x}\| \cos\theta$ when $\|u\| = 1$. Basically this means that for unit-length vectors, when dot products are large, the angle between the vectors must be small—i.e., the vectors must point in the same direction.

Now consider this game, played between two players A and B. Player A provides examples $\mathbf{x}_1, \ldots, \mathbf{x}_T$ for each round (chosen arbitrarily).

In each round, B first makes a prediction $\hat{y}_i$ (say, using a learner L). Then A picks a label y_i, arbitrarily, to assign to $\mathbf{x}_i$. If $\text{sign}(y_i) \neq \text{sign}(\hat{y}_i)$...or if you prefer, if $y_i \hat{y}_i < 0$...then B has made a *mistake*. A is trying to force B to make as many mistakes as possible.

To make this reasonable, we need a few more constraints on what A is allowed to do, and what B will do.

3.2 The perceptron game

Here's one version of the online-learning game. There are three extra rules, to make the game "fair" for B.

Margin γ. A must provide examples that can be separated with some vector $\mathbf{u}$ with margin $\gamma > 0$, ie

$$\exists \mathbf{u} : \forall (\mathbf{x}_i, y_i) \text{ given by } A, \ (\mathbf{u} \cdot \mathbf{x}) y_i > \gamma$$

and furthermore, $\|\mathbf{u}\| = 1$.

To make sense of this, recall that if θ is the angle between $\mathbf{x}$ and $\mathbf{u}$, then

$$\cos\theta = \frac{\mathbf{x} \cdot \mathbf{u}}{\|\mathbf{x}\|\|\mathbf{u}\|}$$

so if $\|\mathbf{u}\| = 1$ then $\|\mathbf{x}\| \cos\theta = \mathbf{x} \cdot \mathbf{u}$. In other words, $\mathbf{x} \cdot \mathbf{u}$ is exactly the result of projecting $\mathbf{x}$ onto vector $\mathbf{u}$, and $\mathbf{x} \cdot \mathbf{u} > \gamma$ means that $\mathbf{x}$ is distance γ away from the hyperplane $\mathbf{h}$ that is perpendicular to $\mathbf{u}$. This hyperplane $\mathbf{h}$ is the separating hyperplane. Notice that $y(\mathbf{x} \cdot \mathbf{u}) > \gamma$ means that $\mathbf{x}$ is distance γ away from $\mathbf{h}$ on the "correct" side of $\mathbf{h}$.

Radius R. A must provide examples "near the origin", ie

$$\forall \mathbf{x}_i \text{ given by } A, \ \|\mathbf{x}\|^2 < R$$

B's strategy. B uses this learning strategy.

1. B's initial guess is $\mathbf{v}_0 = \mathbf{0}$.

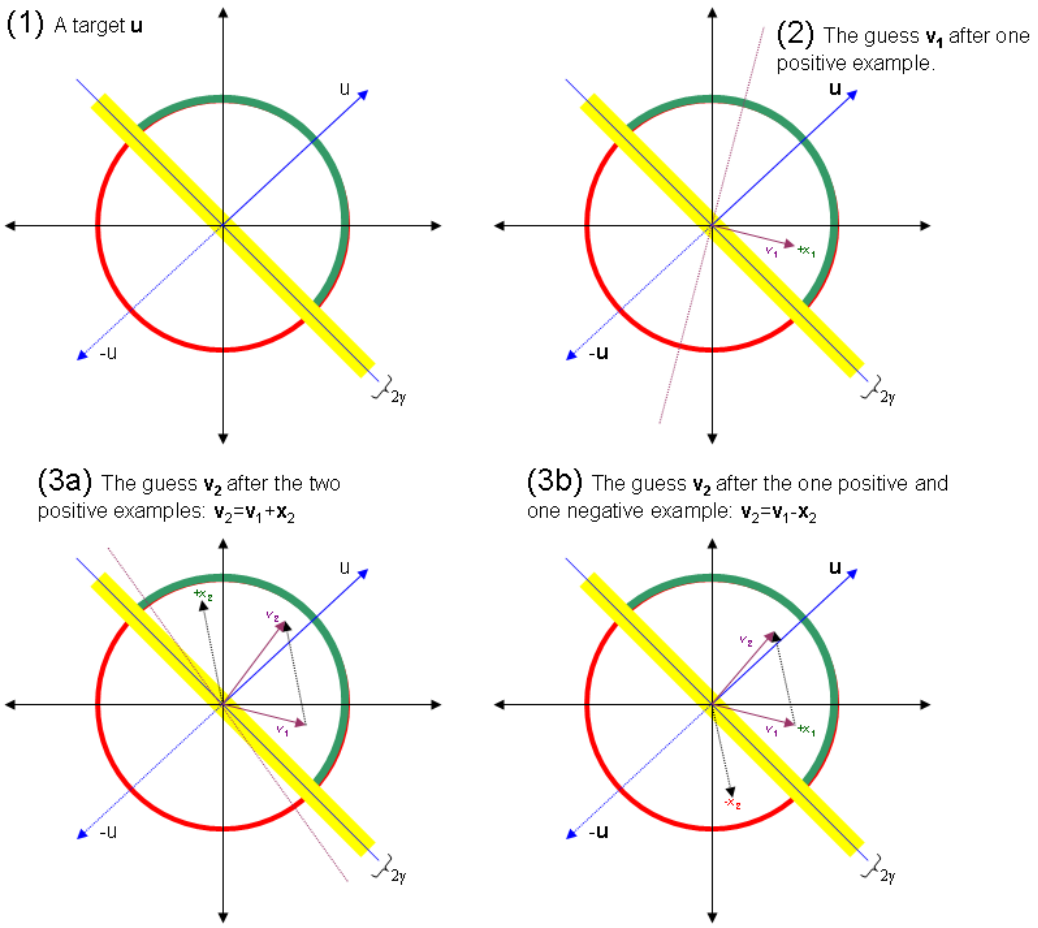

Figure 2: A couple of rounds of the perceptron game, played in two dimensions.

2. At every round, B's prediction is $\text{sign}(\mathbf{v}_k \cdot \mathbf{x}_i)$, where $\mathbf{v}_k$ is the current guess.
3. When B makes a mistake (which we can write concisely as follows: $(\mathbf{v}_k \cdot \mathbf{x}_i)y_i < 0$) then B updates the guess as follows:

$$\mathbf{v}_{k+1} = \mathbf{v}_k + y_i \mathbf{x}_i$$

3.3 An exercise

Don't turn this in, but do try this at home. Play the perceptron game with yourself or a friend in two dimensions on paper or a whiteboard. Start out by drawing $\mathbf{u}$ and the corresponding hyperplane, the separating margin γ, and the radius R. Then draw each $\mathbf{v}_k$. After each mistake, measure $\mathbf{v}_k \cdot \mathbf{u}$ visually by projecting $\mathbf{v}_k$ onto $\mathbf{u}$.

Figure 2 has some examples I did.

3.4 The mistake bound

If you do follow the rules, this simple algorithm is simple to analyze. Amazingly this doesn't depend at all on the dimension of the $\mathbf{x}$'s, and the proof is very simple.

Theorem 1 *Under these rules, it is always the case that $k < R/\gamma^2$. In other words, the number of B's mistakes is bounded by R, the radius the examples fall into, and $1/\gamma^2$, the square of the inverse of the margin.*

The trick is to show two things: that $\mathbf{v}_k \cdot \mathbf{u}$ grows, but $\|\mathbf{v}_k\|$ stays small.

Lemma 1 *$\forall k$, $\mathbf{v}_k \cdot \mathbf{u} \geq k\gamma$. In other words, the dot product between $\mathbf{v}_k$ and $\mathbf{u}$ increases with each mistake, at a rate depending on the margin γ.*

Proof:

$$\begin{aligned} & \mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot \mathbf{u} \\ \Rightarrow \quad & \mathbf{v}_{k+1} \cdot \mathbf{u} = (\mathbf{v}_k \cdot \mathbf{u}) + y_i(\mathbf{x}_i \cdot \mathbf{u}) \\ \Rightarrow \quad & \mathbf{v}_{k+1} \cdot \mathbf{u} \geq \mathbf{v}_k \cdot \mathbf{u} + \gamma \\ \Rightarrow \quad & \mathbf{v}_k \cdot \mathbf{u} \geq k\gamma \end{aligned}$$

The first step is natural to consider—we're just taking the definition of $\mathbf{v}_{k+1}$, and since we care about the dot product with $\mathbf{u}$, we dot-product both sides with $\mathbf{u}$ and then grind away. These steps follow from "B's strategy"; distributivity; the "margin γ" assumption; and then induction; respectively. That's the first lemma.

Now, $\mathbf{v}_k \cdot \mathbf{u}$ could grow even if $\mathbf{v}_k$ points "away from" $\mathbf{u}$—i.e., the angle between them stays large—if $\|\mathbf{v}_k\|$ grows as well. Again, let's start with the definition of $\mathbf{v}_{k+1}$ and "square" each side (in the dot-product sense) to see what happens to the norm.

Lemma 2 *$\forall k$, $\|\mathbf{v}_k\|^2 \leq kR$. In other words, the norm of $\mathbf{v}_k$ grows "slowly", at a rate depending on R.*

Proof:

$$\begin{aligned} & \mathbf{v}_{k+1} \cdot \mathbf{v}_{k+1} = (\mathbf{v}_k + y_i \mathbf{x}_i) \cdot (\mathbf{v}_k + y_i \mathbf{x}_i) \\ \Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_k\|^2 + 2y_i \mathbf{x}_i \cdot \mathbf{v}_k + y_i^2 \|\mathbf{x}_i\|^2 \\ \Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 = \|\mathbf{v}_k\|^2 + [\text{something negative}] + 1\|\mathbf{x}_i\|^2 \\ \Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_k\|^2 + \|\mathbf{x}_i\|^2 \\ \Rightarrow \quad & \|\mathbf{v}_{k+1}\|^2 \leq \|\mathbf{v}_k\|^2 + R \\ \Rightarrow \quad & \|\mathbf{v}_k\|^2 \leq kR \end{aligned}$$

These steps follow from: "B's strategy"; distributivity; "B's strategy" again (in particular, the fact that there was a mistake, and the fact that $y_i \in \{-1, +1\}$); a simplification; the "radius R" assumption; and induction.

Finally, we can wrap this up and prove the theorem. Notice that Lemma 2 bounds the norm of $\mathbf{v}_k$, while Lemma 1 bounds the dot product of $\mathbf{v}_k$ and $\mathbf{u}$. Think about this a second:

if the norm of $\mathbf{v}_k$ is small and the dot product with $\mathbf{u}$ is large, then $\mathbf{v}_k$ might be converging to $\mathbf{u}$—but it depends on the details of how fast these different quantities grow. So the question is, how fast two these trends go? How do we compare them?

It's not obvious, but the way to combine these is to start by squaring the inequality from Lemma 1:

$$\begin{aligned} & (k\gamma)^2 \leq (\mathbf{v}_k \cdot \mathbf{u})^2 \\ \Rightarrow \quad & k^2\gamma^2 \leq \|\mathbf{v}_k\|^2 \|\mathbf{u}\|^2 \\ \Rightarrow \quad & k^2\gamma^2 \leq \|\mathbf{v}_k\|^2 \end{aligned}$$

The last step follows since conveniently, $\|u\| = 1$. Now we have a bound on k^2 in terms of the norm of $\mathbf{v}_k$. Now put this together with Lemma 2, which has a bound on $\mathbf{v}_k$ in terms of kR.

$$k^2\gamma^2 \leq \|\mathbf{v}_k\|^2 \leq kR$$

How sweet—now we have a quadratic function in k that's upper-bounded by a linear function in k. That certainly ought to lead to a bound on k, right? To follow through...

$$\begin{aligned} \Rightarrow \quad & k^2\gamma^2 \leq kR \\ \Rightarrow \quad & k\gamma^2 \leq R \\ \Rightarrow \quad & k \leq \tfrac{R}{\gamma^2} \end{aligned}$$

And that's the whole proof for the mistake bound.

3.5 From on-line to "batch" learning

Interestingly, we've shown that B won't make too many mistakes, but we haven't said anything about how accurate the "last" guess $v_{k_{max}}$ is on i.i.d data. When the data is separable, it seems like it should be ok—but the mistake-bound analysis can be extended to handle non-separable data as well, and in that case it's not at all clear what we should do.

Imagine we run the on-line perceptron and see this result.

i	guess	input	result
1	$\mathbf{v}_0$	$\mathbf{x}_1$	X (a mistake)
2	$\mathbf{v}_1$	$\mathbf{x}_2$	$\surd$ (correct!)
3	$\mathbf{v}_1$	$\mathbf{x}_3$	$\surd$
4	$\mathbf{v}_1$	$\mathbf{x}_4$	X (a mistake)
5	$\mathbf{v}_2$	$\mathbf{x}_5$	$\surd$
6	$\mathbf{v}_2$	$\mathbf{x}_6$	$\surd$
7	$\mathbf{v}_2$	$\mathbf{x}_7$	$\surd$
8	$\mathbf{v}_2$	$\mathbf{x}_8$	X
9	$\mathbf{v}_3$	$\mathbf{x}_9$	$\surd$
10	$\mathbf{v}_3$	$\mathbf{x}_{10}$	X

Our final hypothesis could be the most accurate guess constructed so far, which looks to be $\mathbf{v}_2$ (which was right 3 times and before it had an error, for an empirical error rate of 25%). Or we could use the last guess $\mathbf{v}_4$, which we haven't tested at all...

Freund and Schapire suggest using a mixture of all of these, weighting them by their accuracy on the data—or more precisely, by m_k, the number of examples they were used for. In this case, we'd randomly pick $\mathbf{v}_0$ with probability 1/10, $\mathbf{v}_1$ with probability 3/10, $\mathbf{v}_2$ with probability 4/10, and $\mathbf{v}_3$ with probability 2/10. After picking our $\mathbf{v}_k$, we just use its prediction.

Each $\mathbf{v}_k$'s empirical error is $1/m_k$, so if we let $m = \sum m_k$ then the probability of an error using this voting scheme is

$$\begin{aligned} P(\text{error in } \mathbf{x}) &= \sum_k P(\text{error on } \mathbf{x} | \text{picked } \mathbf{v}_k) P(\text{picked } \mathbf{v}_k) \\ &= \sum_k \frac{1}{m_k} \frac{m_k}{m} = \sum_k \frac{1}{m} = \frac{k}{m} \end{aligned}$$

This is easy to understand and seems to work well experimentally.

Something even easier is implement is approximate the voting scheme with a weighted average, and classify with a single vector

$$\mathbf{v}_* = \sum_k (\frac{m_k}{m} \mathbf{v}_k)$$

3.6 Putting it together

Some practical tricks: usually when you're using the voted perceptron in batch mode, you run it as if you were learning on-line, but make several passes over the data. (Once is ok, but if you're not in a big hurry, five might be better). With data that is preprocessed so that it is binary, and has that special x_0 feature that is always true, the averaged perceptron is often a very accurate learner, and is amazingly robust to high dimensions, as well as potential problems like redundant features, non-separable examples. Its performance is (not by accident) often quite similar to the performance of support vector machines, a popular but much more CPU-intensive learning method. And the best part is that it is incredibly simple to implement—especially in a language like Matlab that supports sparse vector operations. Here's the outline of the code:

1. set $\mathbf{v} = \mathbf{w} = \mathbf{0}$
2. for $k = 1, \ldots, K$ *—i.e., make K passes over the data*
   (a) for $t = 1, \ldots, m$ *—i.e.,look at each example* $\langle \mathbf{x}_t, y_t \rangle$
      i. if $(\mathbf{x}_t \cdot \mathbf{v} y_t < 0)$ then set $\mathbf{v} = \mathbf{v} + y_t \mathbf{x}_t$
      ii. set $\mathbf{w} = \mathbf{w} + \mathbf{v}$
3. set $\mathbf{w} = \frac{1}{Km} \mathbf{w}$ *—now* $\mathbf{w}$ *is the weighted average of the* $\mathbf{v}$*'s*
4. return the linear classifier $\hat{y} = \mathbf{w}\mathbf{x}$