Hyperplanes

Alex Sverdlov

alex@theparticle.com

1 Hyperplanes

A hyperplane is a fancy name for a 'plane' in N-dimensions: this is just a line in 2D, and a plane in 3D as illustrated in Figure 1.

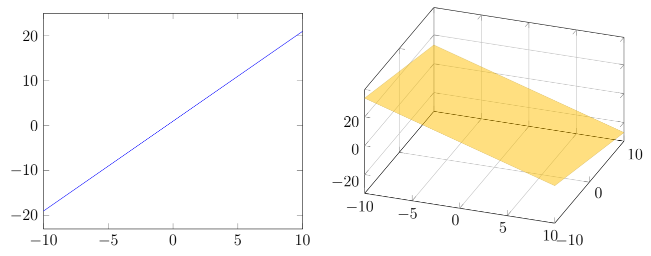

Figure 1: A line in 2D: $Ax + By = D$, and a plane in 3D: $Ax + By + Cz = D$

This can be extended indefinitely. To avoid running out of variables, we often write the plane as:

$$w_1x_1 + w_2x_2 + \cdots + w_nx_n = D$$

and to avoid that awkward D at the end, we often create $w_0 = -D$ and always set $x_0 = 1$. That way the whole thing becomes:

$$w_0x_0 + w_1x_1 + \cdots + w_nx_n = 0 \qquad \text{in vector notation: } \boldsymbol{w}^T\boldsymbol{x}$$

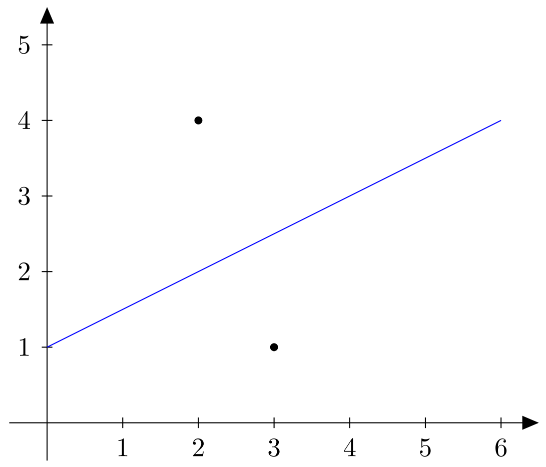

Figure 2: Dividing Line: $-x + 2y - 2 = 0$.

To check if a point $\boldsymbol{x}$ is exactly on the hyperplane we check $\boldsymbol{w}^T\boldsymbol{x} = 0$. To check if the point is in *front* of the plane, we check for $\boldsymbol{w}^T\boldsymbol{x} > 0$, and *back* of the plane as $\boldsymbol{w}^T\boldsymbol{x} < 0$.

For example, Figure 2, we have: $-x + 2y - 2 = 0$. That is, $w_1 = -1$, $w_2 = 2$, and $w_0 = -2$. Testing point $(2, 4)$ against it we get $-(2) + 2(4) - 2 > 0$, or *front* of the line. That is, $x_1 = 2$, $x_2 = 4$, and we always set $x_0 = 1$. Doing an inner product $\boldsymbol{w}^T\boldsymbol{x} > 0$. The second point $(3, 1)$ is checked the same exact way:

$$[w_0, w_1, w_2] \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = [-2, -1, 2] \times \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} < 0$$

Because $\boldsymbol{w}^T\boldsymbol{x}$ (or $(-2 \times 1) + (-1 \times 3) + (2 \times 1)$) is less then zero, we know $(3, 1)$ is on the *back* of the line. This is really the power of hyperplanes—they are a convenient linear modeling tool. The notion of *front* vs *back* is arbitrary (we can always flip the 'direction' of a line by multiplying all weights by -1).

1.1 Perceptrons

Perceptrons are literally hyperplanes, oriented in any direction. This power comes at a cost of readability—unlike decision trees, it is very hard to figure out what, if any, meaning exists in a hyperplane.

The idea of artificial neurons dates back to McCulloch & Pitts (1943) [**?**], when they proposed using an artificial neuron for computation—defining a mathematical abstraction of a biologically inspired neuron.

The McCulloch & Pitts neuron contains a set of real valued weights $(w_1, \ldots, w_n)$ and accepts $(x_1, \ldots, x_n)$ as input (without the $x_0 = 1$, that is handled separately by *threadhold*). The function it applies is:

$$o(\boldsymbol{x}) = step(\boldsymbol{x}^T \boldsymbol{w}) \qquad\qquad step(x) = \begin{cases} 1 & \text{if } y \geq threadhold \\ 0 & \text{otherwise} \end{cases}$$

where $\boldsymbol{x}^T \boldsymbol{w}$ is the inner product of $\boldsymbol{x}$ and $\boldsymbol{w}$ column vectors, and *step* a linear step function at *threshold*. The w_i values are normalized to a $(0, 1)$ or $(-1, 1)$ range, and both inputs and outputs are binary. See Figure 3.

Simple as they are, arrangements of such neurons were shown to compute any binary function. The inability to calculate XOR function (famously presented as a major limitation by Minsky & Papert (1969) [**?**]) is only applicable to a single layer of neurons. Since a neuron is a hyperplane, it cannot split XOR since that is not linearly separable, see Figure 4. Layering perceptrons into networks (as McCulloch & Pitts have done in their paper) overcomes this limitation.

McCulloch & Pitts did not define any training method; like programming, one had to adjust the neuron weights to compute different binary functions.

Perceptrons were developed by Frank Rosenblatt in 1958. [**?**] Heavily based on the

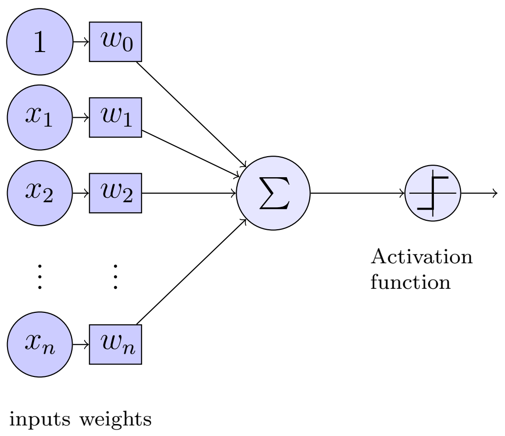

Figure 3: The Perceptron. Thought different models differ in detail, they all follow this general approach. (picture by *m0nhawk* on TEX StackExchange)

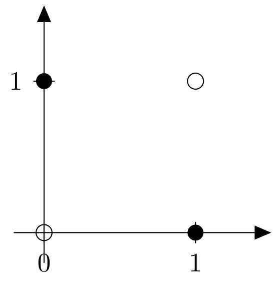

Figure 4: The XOR function. There is no single line that can separate the filled circles from non-filled ones.

McCulloch & Pitts neuron, this model used more flexible weights, and had an adaptive component. A perceptron is a function:

$$o(\boldsymbol{x}) = sign(\boldsymbol{x}^T\boldsymbol{w}) \qquad\qquad sign(x) = \begin{cases} 1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where $\boldsymbol{x}^T\boldsymbol{w}$ is the inner product of input $\boldsymbol{x}$ and weight vector $\boldsymbol{w}$. The first input x_0 is always assumed to have value 1, and weight w_0 is the corresponding threshold.

Training a perceptron is iterative. For every training sample $(\boldsymbol{x}, y)$ we adjust the weights by

$$w_i = w_i + \lambda(y - o(\boldsymbol{x}))x_i$$

The $y - o(\boldsymbol{x})$ gets the classification error (or 0 if no error was made on the training example). The adjustment is then weighted by learning rate parameter λ and input x_i. In other words, if there was an error, and input x_i was tiny, we want to make a tiny adjustment to w_i. If x_i was large, then we want to make a similarly large adjustment to w_i.

For example, suppose the training example has $x = 2$ and $y = 1$, and our perceptron is $w = -1$ with 0 threshold. The $o(x)$ is threfore -1. We need to adjust weight *upwards* (governed by $y - o(x)$, in our example $1 - (-1) = 2$) by some fraction λ (perhaps set to 0.1) of x. Therefore we adjust weight by:

$$\lambda(y - o(\boldsymbol{x}))x_i = 0.1 \times (1 - (-1)) \times 2 = 0.4$$

A limitation of the above learning rule is that there is no notion of good $\boldsymbol{w}$ beyond the correct or incorrect value of 1 or -1, see Figure 5. If the sets are not separable, the perceptron will semi-randomly stumble in adjusting the weight vector until stopped. It is not a robust algorithm by any means, but it was a good start in the right direction.

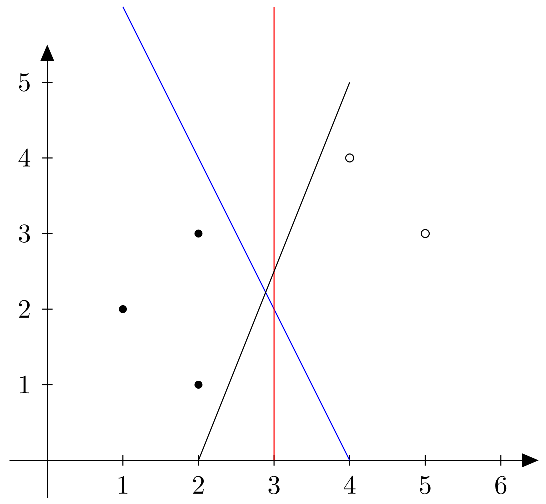

Figure 5: The perceptron learning rule will not adjust weights once they produce the correct classification—in other words, the different lines in this figure are all equally correct.

1.2 Delta Rule

Bernard Widrow (1965) [?] and Ted Hoff came up with a much better way to train the perceptron. Let us start by defining the total error for a neuron with weights $\boldsymbol{w}$:

$$E(\boldsymbol{w}) \equiv \frac{1}{2}\sum_{i\in D}(y_i - \boldsymbol{x}_i^T\boldsymbol{w})^2$$

where D is the set of all training data. The E function is essentially the *sum of squares* of all errors on the dataset D. We can differentiate E with respect to $\boldsymbol{w}$:

$$\Delta E(\boldsymbol{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \qquad \frac{\partial E}{\partial w_i} = \sum_{i\in D}(y_i - \boldsymbol{x}_i^T\boldsymbol{w})(-x_i)$$

The $\Delta E(\boldsymbol{w})$ is the gradient vector, with a component for each weight. We can adjust the weights via:

$$w_i = w_i + -\lambda\Delta E(\boldsymbol{w})$$

where λ is the learning rate. The $\Delta E(\boldsymbol{w})$ points in the direction that *increases* $E(\boldsymbol{w})$, so we need the negative sign to adjust in the decreasing direction. This derivation is adopted from Mitchell. [**?**]

The iterative version of the delta rule is:

$$w_i = w_i + \lambda(y - \boldsymbol{x}^T\boldsymbol{w})x_i$$

This method has a smoother learning behavior than perceptron learning rule, and will continue to adjust $\boldsymbol{w}$ until it reaches an optimum value, even when all the examples are correctly classified.

The function is identical to the non-thresholded Perceptron learning—it is amazing what that small adjustment has done.

1.3 Least Squares

It turns out there is a more direct way to solve the weight adjustment problem. When the target label y is a real number (and $\boldsymbol{x}$s are numeric), the 'labeling' task becomes regression—we are fitting some function to data points. Least squares finds a hyperplane that best fits the $\boldsymbol{x}$s. Our model is:

$$\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$$

where $\boldsymbol{X}$ is a matrix with individual observables $\boldsymbol{x}$ as rows, $\boldsymbol{y}$ is a column vector of target variables. This is the heart of linear algebra. It seems we should be able to solve $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$ for $\boldsymbol{w}$ by solving a system of linear equations (e.g. $\boldsymbol{w} = \boldsymbol{X}^{-1}\boldsymbol{y}$). For example:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 2 \\ 1 & 4 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

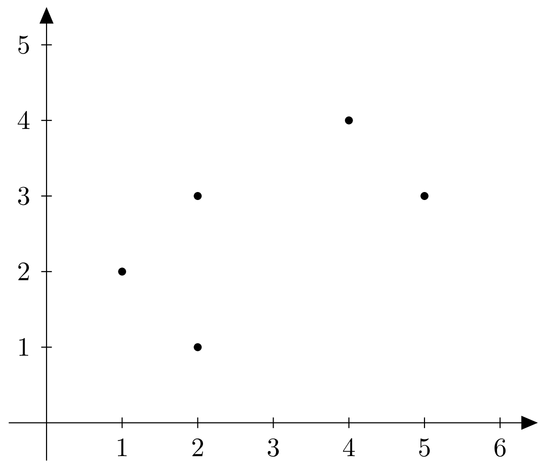

Figure 6: Sample points for Least Squares regression.

We can solve it by inverting $\boldsymbol{X}$, and solving for $\boldsymbol{w}$

$$\boldsymbol{X}^{-1} = \begin{bmatrix} 1 & 2 \\ 1 & 4 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -1 \\ -0.5 & 0.5 \end{bmatrix} \qquad \text{then} \qquad \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -0.5 & 0.5 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}$$

Unfortunately there are usually many more observables than $\boldsymbol{w}$s, such as:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 5 \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 4 \\ 3 \end{bmatrix}$$

Such problems are overdetermined ($\boldsymbol{X}$ is not square), and training data contains noise—we would not be able to fit a line through the data, because no such line exists, see Figure 6.

We can rewrite the problem as: $\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w} = \boldsymbol{\gamma}$, where $\boldsymbol{\gamma}$ is the *error*, either positive or negative. We can square the error to get a positive number, then just as in Delta Rule, minimize the square error. Rewriting the squared error function:

$$\begin{aligned} E(\boldsymbol{w}) = \|\gamma\|^2 &= (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^2 \\ & (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) \\ & (\boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{y} + \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}) \end{aligned}$$

One clever way of finding minimums (or maximums) is to differentiate, then set derivative to zero, and solve. The derivative is:

$$\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} = -2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = 0$$

Which leads to what are called 'normal equations':

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y} \qquad \text{which leads to:} \qquad \boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

If $\boldsymbol{X}^T\boldsymbol{X}$ is invertible, that is it, we can directly solve for $\boldsymbol{w}$. If some columns of $\boldsymbol{X}$ are not independent, then is $\boldsymbol{X}^T\boldsymbol{X}$ not invertible, and we need to make an adjustment. Adding $\lambda\boldsymbol{I}$ to $\boldsymbol{X}^T\boldsymbol{X}$ ensures the inverse always exists. λ here is some tiny number, like 0.001.

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

This is the regression solution for situations when $\boldsymbol{X}$ is $N \times M$ matrix, and N is much *bigger* than M (has many more rows than columns). The $\boldsymbol{X}^T\boldsymbol{X}$ is a square matrix sized $M \times M$. Its running time depends on inverting $\boldsymbol{X}^T\boldsymbol{X}$—the algorithm is very fast for tall and skinny

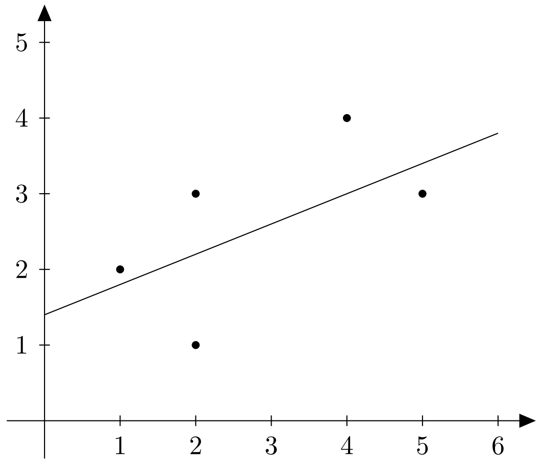

Figure 7: Sample points plotted along with the 'best' line: $y = 0.4x + 1.4$

matrices.

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad \boldsymbol{X}^T\boldsymbol{X} = \begin{bmatrix} 5 & 14 \\ 14 & 50 \end{bmatrix}, \quad (\boldsymbol{X}^T\boldsymbol{X})^{-1} \approx \begin{bmatrix} 0.92 & -0.26 \\ -0.26 & 0.09 \end{bmatrix}$$

Plugging that into $\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$ we get

$$\boldsymbol{w} \approx \begin{bmatrix} 0.92 & -0.26 \\ -0.26 & 0.09 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 4 & 5 \end{bmatrix} \times \begin{bmatrix} 2 \\ 1 \\ 3 \\ 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 1.4 \\ 0.4 \end{bmatrix}$$

This example dataset is probably not the best illustration of line fitting—the points do not appear to be on or near the line at all, see Figure 7 for the plot of $\boldsymbol{w}$. That said, the least squares algorithm found the 'best' line to fit them anyway.

1.4 Least Squares Duality

Now for a bit of magic (Shawe-Taylor & Cristianini (2004) [?] derivation). Using *Sherman-Morrison-Woodbury formula* [?] we can rewrite $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T$ as $\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}$, giving us another way of solving for $\boldsymbol{w}$ [1]:

$$\boldsymbol{w} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

This is the regression solution for situations when $\boldsymbol{X}$ is $N \times M$ matrix, and N is much *smaller* than M (has many more columns than rows). The $\boldsymbol{X}\boldsymbol{X}^T$ is a square matrix sized $N \times N$. Its running time depends on inverting $\boldsymbol{X}\boldsymbol{X}^T$.

Since most datasets have more records than attributes it seems this second derivation does not gain us much. A useful thing to notice is that $\boldsymbol{w}$s are now a linear combination of inputs.

$$\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{\alpha} \qquad\qquad \boldsymbol{\alpha} = (\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}\boldsymbol{y} = (\boldsymbol{G} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y}$$

where $\boldsymbol{G}_{ij} = \boldsymbol{x}_i^T\boldsymbol{x}_j$. To apply this $\boldsymbol{w}$ to a new sample $\boldsymbol{x}$:

$$\boldsymbol{w}^T\boldsymbol{x} = (\boldsymbol{X}^T\boldsymbol{\alpha})^T\boldsymbol{x} = (\boldsymbol{X}^T(\boldsymbol{G} + \lambda\boldsymbol{I})^{-1}\boldsymbol{y})^T\boldsymbol{x} = \boldsymbol{y}^T(\boldsymbol{G} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}\boldsymbol{x} = \boldsymbol{\alpha}\boldsymbol{k}$$

where $\boldsymbol{k}$ is a vector where each $k_i = \boldsymbol{X}_i^T\boldsymbol{x}$.

[1] This does not imply $(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T$ is equal to $\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})^{-1}$; pseudo-inverses are not exact.

Using our original example matrix, the $\boldsymbol{X}\boldsymbol{X}^T$ is much bigger than $\boldsymbol{X}^T\boldsymbol{X}$ for this problem:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 2 \\ 1 & 4 \\ 1 & 5 \end{bmatrix}, \quad \boldsymbol{X}\boldsymbol{X}^T = \begin{bmatrix} 2 & 3 & 3 & 5 & 6 \\ 3 & 5 & 5 & 9 & 11 \\ 3 & 5 & 5 & 9 & 11 \\ 5 & 9 & 9 & 17 & 21 \\ 6 & 11 & 11 & 21 & 26 \end{bmatrix}$$

Not surprisingly, $\boldsymbol{X}\boldsymbol{X}^T$ is not invertible (2nd and 3rd rows are the same). This is where the $\lambda\boldsymbol{I}$ adjustmenet becomes important. The $(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I})$ is definitely invertible.

1.5 Discriminators

While the 'least squares' method described above is used primarily for interpolation and extrapolation, a similar technique can be used for classification. [?] The idea is to find the hyperplane that *splits* the provided training data.

Given a training set:

$$\boldsymbol{X} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_L, y_L)\}$$

where $y_i \in \{-1, +1\}$ indicates the class, we will use $\boldsymbol{X}_+$ as shorthand for all the positive training cases, and similarly $\boldsymbol{X}_-$ for all the negative ones. Our model is a hyperplane, with weights $\boldsymbol{w}$, and distance D, such that:

$$w_1x_1 + \cdots + w_Nx_N = D$$

With such a hyperplane, we get a notion of things being in 'front' of the plane and in the 'back' of the plane. If we plug $\boldsymbol{x}$ into the plane equation (represented by $\boldsymbol{w}$ and D), and get a positive value, then $\boldsymbol{x}$ is in front of the plane, etc. For this section, we will use the same

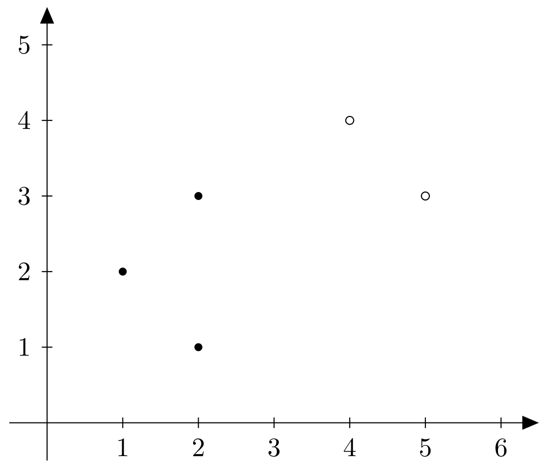

Figure 8: Example data for classification.

dataset as before, Figure 8.

$$\boldsymbol{X}_+ = \begin{bmatrix} 1 & 2 & +1 \\ 2 & 1 & +1 \\ 2 & 3 & +1 \end{bmatrix}, \quad \boldsymbol{X}_- = \begin{bmatrix} 4 & 4 & -1 \\ 5 & 3 & -1 \end{bmatrix},$$

1.5.1 Fisher's Linear Discriminant

Perhaps the simplest idea for a classifier is to calculate means of positive and negative examples: $\boldsymbol{\mu}_+$ and $\boldsymbol{\mu}_-$, then create a vector from one to the other and normalize:

$$\boldsymbol{w} = \frac{\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-}{\|\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-\|}$$

The D can be used to place the plane right between the two means, e.g.

$$D = \boldsymbol{w}\frac{1}{2}(\boldsymbol{\mu}_+ + \boldsymbol{\mu}_-)$$

This simple method is a special case of Fisher's Linear Discriminant (1936) [**?**], and it works when covariance matrices for $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ are mostly multiples of identify matrix: both $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ are spheres around their respective means, with very little skew. When this is not the case, we need to incorporate the covariance matrices ($\boldsymbol{\Sigma}_+$ and $\boldsymbol{\Sigma}_-$) into the calculation:

$$\boldsymbol{w} = (\boldsymbol{\Sigma}_+ + \boldsymbol{\Sigma}_-)^{-1}(\boldsymbol{\mu}_+ - \boldsymbol{\mu}_-)$$

We are still creating a hyperplane from one mean to the next—we just twist it by the inverse of covariance matrices. For example, if $\boldsymbol{X}_+$ is particularly stretched out in direction i then the resulting $\boldsymbol{w}$ will be more orthogonal in that direction i. The D for the hyperplane can be chosen using the same method as above—or we can scale it by standard deviation away from each class (for situations when deviations of $\boldsymbol{X}_+$ and $\boldsymbol{X}_-$ are not the same).

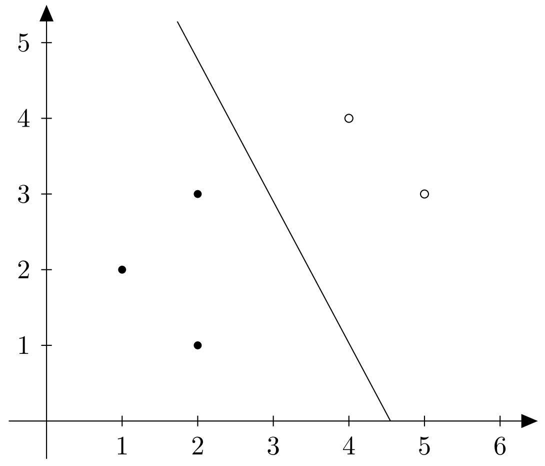

Figure 9: Fisher's Linear Discriminant, $0.88 * x + 0.47 * y - 4 = 0$

1.6 Maximal Margin Separator

When we apply the Fisher's Linear Discriminant ($0.88 * x + 0.47 * y - 4$) to the two closest points, $(2, 3)$ and $(4, 4)$ we get -0.83 and 1.4 respectively. They are correctly classified as far as their sign is concerned, but their scale is different—if each is the closest point to the hyperplane, what makes one more positive than the other negative? In other words, had we used just the closest points to build a linear discriminant we would get a different classifier.

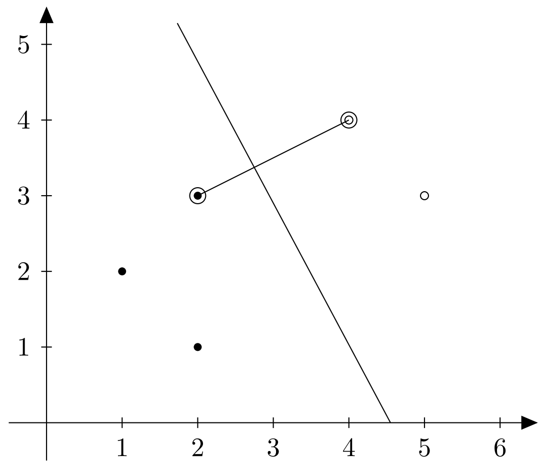

Figure 10: The two closest points to Fisher's Linear Discriminant are not the same distance to the separating hyperplane.

The maximal margin separator is essentially the idea that to achieve maximum generalizability, the separating surfaces should maximize distance to both negative and positive examples.

Let us start with the linear hyperplane model, $\boldsymbol{w}^T\boldsymbol{x} = D$, and assume that $\boldsymbol{w}$ is normalized (has a norm of 1). We can rewrite this as: $\boldsymbol{w}^T\boldsymbol{x} - D = 0$. For all $\boldsymbol{x}$ that are on the hyperplane, this model will produce 0.

For other $\boldsymbol{x}$ that are not on the hyperplane, this model produces the *distance* from the hyperplane to $\boldsymbol{x}$. We would like to maximize this distance, provided $\boldsymbol{x}$ is on the correct side of the hyperplane. In other words, maximize $\boldsymbol{w}^T\boldsymbol{x} - D$ subject to $y_i(\boldsymbol{w}^T\boldsymbol{x}_i - D) \geq 1$ for all training data i.

Suppose the maximum of $\boldsymbol{w}^T\boldsymbol{x} - D = \gamma$, we can then divide out the γ:

$$\frac{\boldsymbol{w}^T\boldsymbol{x}}{\gamma} - \frac{D}{\gamma} = \frac{\gamma}{\gamma} = 1$$

In other words, we can turn the maximization of $\boldsymbol{w}^T\boldsymbol{x} - D$ problem into the minimization of $\|\boldsymbol{w}\|$ problem, subject to same constraints. Note that here $\boldsymbol{w}$ is not normalized.

A way to solve such optimization problems is to use Lagrange multipliers. We rewrite the optimization problem as:

$$f(\boldsymbol{w}, D) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_i \alpha_i[y_i(\boldsymbol{w}^T\boldsymbol{x}_i - D) - 1]$$

Take derivatives, set to zero, and solve for $\boldsymbol{w}$:

$$\begin{array}{lll}\frac{\partial f(\boldsymbol{w},D)}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_i \alpha_i y_i \boldsymbol{x}_i = 0 & \Rightarrow & \boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x}_i \\ \frac{\partial f(\boldsymbol{w},D)}{\partial D} = \sum_i \alpha_i y_i = 0 & & \end{array}$$

The problem of course is that we need to solve for $\boldsymbol{\alpha}$ before we solve for $\boldsymbol{w}$. However, now that we have a solution for $\boldsymbol{w}$ we can just plug into the original formula and simplify:

$$\begin{aligned} f(\boldsymbol{\alpha}) &= \tfrac{1}{2}\left(\textstyle\sum_i \alpha_i y_i \boldsymbol{x}_i\right)^T \left(\textstyle\sum_j \alpha_j y_j \boldsymbol{x}_j\right) - \textstyle\sum_i \alpha_i \left[y_i \left(\left(\textstyle\sum_j \alpha_j y_j \boldsymbol{x}_j\right)^T \boldsymbol{x}_i - D\right) - 1\right] \\ &= \tfrac{1}{2}\textstyle\sum_i\sum_j \alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j - \left[\textstyle\sum_i\sum_j \alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j - 0 - \textstyle\sum_i \alpha_i\right] \\ &= \textstyle\sum_i \alpha_i - \tfrac{1}{2}\textstyle\sum_i\sum_j \alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j \end{aligned}$$

Flipping the sign, we aim to minimize $f(\boldsymbol{\alpha})$:

$$\min_{\boldsymbol{\alpha}} f(\boldsymbol{\alpha}) = \frac{1}{2}\sum_i\sum_j \alpha_i\alpha_j y_i y_j \boldsymbol{x}_i^T\boldsymbol{x}_j - \sum_i \alpha_i$$

subject to $\alpha_i \geq 0$ for all i, and $\sum_i \alpha_i y_i = 0$. This is a simple Quadratic Programming problem that can be solved by setting up a linear KKT[2] system:

$$\left[\begin{array}{c|c} 0 & \boldsymbol{y}^T \\ \hline \boldsymbol{y} & \boldsymbol{H} \end{array}\right] \left[\begin{array}{c} -D \\ \hline \boldsymbol{\alpha} \end{array}\right] = \left[\begin{array}{c} 0 \\ \hline \mathbf{1} \end{array}\right]$$

where $\boldsymbol{H}_{ij} = y_i y_j \boldsymbol{x}_i \boldsymbol{x}_j$, $\boldsymbol{y} = (y_1, \ldots, y_L)$, $\mathbf{1} = (1_1, \ldots, 1_L)$. $\boldsymbol{H}$ is what is known as a Hessian matrix, and $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_L)$ are Lagrange multipliers from the dual solution. Note that we can directly solve for $\boldsymbol{\alpha}$ and D (via least squares). We can then solve for $\boldsymbol{w}$ via:

$$\boldsymbol{w} = \boldsymbol{X}^T \left[\boldsymbol{\alpha} \times \boldsymbol{y}\right]$$

where $\boldsymbol{\alpha} \times \boldsymbol{y}$ is an element-wise multiplication. This $\boldsymbol{w}$ is not yet the maximal margin separator—next section on sparseness addresses the maximal margin.

1.7 Sparseness & Support Vector Machines

Above we saw that we can build regression and classification models using robust solutions, grounded firmly in linear algebra methods. The problem of course is the size of the $\boldsymbol{G}$ or $\boldsymbol{H}$ matrix. If we have a moderate number of samples, perhaps 10000, then we are talking about a non-sparse 10000×10000 matrix—and inverting something like that is a challenge—if we have a million samples, the whole solution becomes impractical.

A key insight comes from the $\boldsymbol{\alpha}$ vector. This is really what we are optimizing. Perhaps we can avoid solving for all of them at the same time? This is exactly the thinking behind Support Vector Machines, first proposed by Vapnik (1963) [**?**, **?**] and later developed by many others.

The SVM algorithms iteratively solves for some subset of $\boldsymbol{\alpha}$s, and continue to iterate

[2]Karush-Kuhn-Tucker

until all the $\boldsymbol{\alpha}$s satisfy the KKT. At this point, for most practical problems, only a tiny subset of the $\boldsymbol{\alpha}$s will be non-zero—meaning that only a tiny subset of the whole dataset is contributing to the $\boldsymbol{w}$. It also makes kernel appliation (Section 1.10) practical. Figure 11 shows an example of SVM for the example dataset.

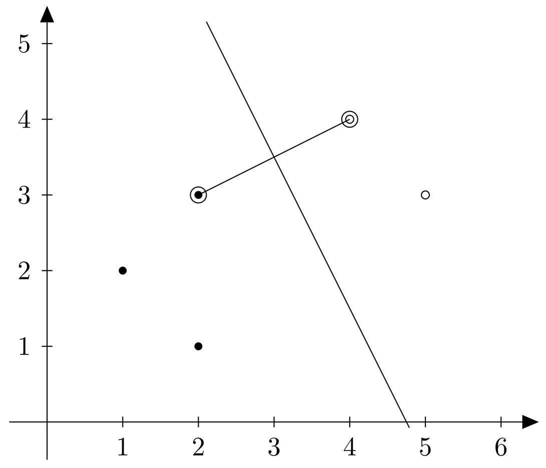

Figure 11: The SVM discriminator: $0.89443 * x + 0.44721 * y - 4.2485$, The $(2, 3)$ and $(4, 4)$ samples have $\alpha_i = 1$, the rest of the $\boldsymbol{\alpha}$ vector is 0.

The mechanical question then becomes how to setup the iteration to efficiently clamp down the value of most unimportant αs to zero. Practical considerations are: the iteration has a matrix inversion as the inner loop—the more α values we try to solve for, the more complicated this inner loop becomes.

Vapnik proposed a "chunking" algorithm. The key idea is that rows/columns with corresponding $\alpha_i = 0$ are irrelevant and can be skipped. Each iteration begins by collecting all non-zero Lagrange multipliers from last step and the M worst examples that violate KKT conditions (for some value of M). Since there is no hard control on how many non-zero Lagrange multipliers may exist from iteration to iteration, this solution has unpredictable runtime performance.

This was later improved by Osuna (1997) [**?**] who proposed to have a constant size matrix. Every iteration would operate on the same number of Lagrange multipliers. Osuma also proved that a large QP problem can be broken down into a series of smaller QP problems. As long as at least one example violates KKT conditions, then the overall problem will converge. Because the size of each sub-problem is fixed and limited, this solution can work on arbitrary large inputs—each subproblem doing a fixed amount of work.

Osuna's solution led Platt (1998) [?] to develop the Sequential Minimal Optimization (SMO) algorithm, that uses just two Lagrange multipliers per iteration. It turned out that this can be solved analytically, avoiding the whole QP optimization as an inner loop, and is the fastest way of doing general SVMs right now.

Smola (2004) [?, ?] applied SMO ideas to regression—the Lagrange multipliers are bounded to a certain margin around the hyperplane. Joachims (2006) [?] has developed Boosting-like methods can be used to train linear SVMs in linear time. Syed (1999) [?] developed methods to train SVMs incrementally.

1.8 Non-Linear Embedding

If we wanted to fit an exponential function, none of the above mentioned linear methods would work. One easy tweak we could do is 'embed' our linear data in non-linear space. We can do this by defining a non-linear function Φ, and transforming our data using that function. In other words, instead of working with $\boldsymbol{x}$, we would work with $\Phi(\boldsymbol{x})$.

The function Φ can be anything at all. It can reduce or increase the dimensionality of the sample point $\boldsymbol{x}$. For example, a 2-dimensional $\boldsymbol{x}$ may be turned into a 3-dimensional $\Phi(x)$:

$$\Phi(x_1, x_2) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

This has the capacity of turning our 'learning lines' method into a 'learning curves' method.

To see why this works, consider fitting points to $y = Be^{Ax}$. We can take log of both sides to get $\ln(y) = Ax + \ln(B)$, which is linear. The Φ embedding would apply the ln function to Y, and upon output, apply exponential to get B.

Similarly, to fit power function $y = B * x^a$ we take log of both sides to get $\ln(y) = \ln(B) + a * \ln(x)$, which is now also linear.

To fit polynomials we "embed" higher dimensions that are powers of x. For example, instead of

$$\begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 1 & 5 \\ 1 & 7 \\ 1 & 11 \\ 1 & 13 \end{bmatrix}$$

which would fit a line, we can fit a 3rd degree polynomial just by tweaking that matrix to be:

$$\begin{bmatrix} 1 & 2 & 2^2 & 2^3 \\ 1 & 3 & 3^2 & 3^3 \\ 1 & 5 & 5^2 & 5^3 \\ 1 & 7 & 7^2 & 7^3 \\ 1 & 11 & 11^2 & 11^3 \\ 1 & 13 & 13^2 & 13^3 \end{bmatrix}$$

The resulting solutions will have the form $y = D + Cx + Bx^2 + Ax^3$. This can be extended to any degree polynomial you care to fit.

1.9 Logistic Regression

Logistic regression is a form of non-linear Φ embedding discussed above. Often we have problems that require learning and extrapolating probabilities—and those are always in the

0 to 1 range. To use a linear model, we need to project that 0 to 1 range onto the $-\infty$ to $+\infty$ range. We do this via the logit function

$$\mathrm{logit}(x) = \log\left(\frac{x}{1-x}\right)$$

The logit function is the log of the Odds ratio, and is illustrated in Figure 12. Notice that 'probabilities' that are very close to 1, will get an extremely high y value (we need to clamp it at some high value, since ∞ is kind of hard to represent on a computer), and vice versa.

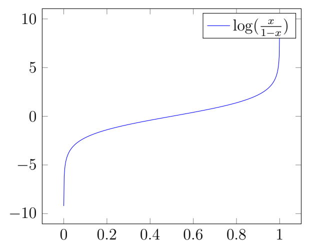

Figure 12: The logit function.

The embedding transforms the $\boldsymbol{y}$ target probability into $\mathrm{logit}(\boldsymbol{y})$. The model is trained on the transformed values. Once we have our linear model, the outputs of $\boldsymbol{w}^T\boldsymbol{x}$ will be linear and in $-\infty$ to $+\infty$ range. We need to turn those into probabilities—in other words, we need the inverse of the logit function, which happens to be the sigmoid function, Figure 13:

$$\mathrm{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

The output of logistic regression is determined via $\mathrm{sigmoid}(\boldsymbol{w}^T\boldsymbol{x})$ and it is always a value between 0 and 1; something that can be interpreted as 'probability'.

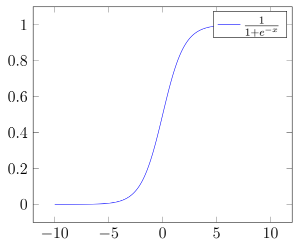

Figure 13: The sigmoid function.

1.10 Kernels

Kernels are just a clever method to do Φ embedding, utilizing the $\boldsymbol{G}$ matrix computation pipeline, without actually computing the Φ embedding. For example (due to [**?**]):

$$\begin{aligned} K(\Phi(\boldsymbol{x}), \Phi(\boldsymbol{z})) &= (x_1^2, x_2^2, \sqrt{2}x_1x_2)^T(z_1^2, z_2^2, \sqrt{2}z_1z_2) \\ &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1x_2z_1z_2 \\ &= (x_1z_1 + x_2z_2)^2 \\ &= (\boldsymbol{x}^T\boldsymbol{z})^2 \end{aligned}$$

In other words, we can avoid a lot of calculation by simply squaring the elements of the $\boldsymbol{G}$ matrix. There are many more of these kernels—some even embed the data into an infinite dimensional space, such as the Gaussian kernel—this would be impossible to calculate without using this kernel trick.

Kernels have become a key piece in algorithm creation—regression and classification are domain and data independent: they will work on any data in any domain. Kernels on the other hand are often crafted along with data preparation.

Kernels take any two data examples, and produce what amounts to as a similarity score. This could be calculated via an algorithm, heuristically assigned, etc. For text data, this could be counting words. For image data, comparing histograms, etc.