CHAPTER

4 Linear classification

We next turn our attention to using linear models for classification, in which we are required to make a discrete prediction (or decision) about a data point given its features x. A linear classifier is one whose decision function is a linear function of the input features; so, for a binary classifier between classes $\{+1, -1\}$ ("positive" versus "negative"), we can write a decicion function as

$$\hat{y} = f(x\,;\,\theta) = \text{sign}(\theta \odot x^T) = \begin{cases} +1 & \theta \odot x^T > 0 \\ -1 & \text{otherwise} \end{cases} \tag{4.1}$$

where, as in linear regression, we take $x = [1, x_1, x_2, \ldots, x_n]$ to be our usual vector of feature values (with constant feature $x_0 = 1$), and $\theta = [\theta_0, \theta_1, \ldots \theta_n]$ is a vector of weights. Note that, for binary classes $\{+1, -1\}$, this is simply the sign of the linear response $r = \theta \odot x^T$; if we were to instead select our binary classes to be $\{1, 0\}$, we could alternatively define

$$\hat{y} = f(x; \theta) = \mathbb{1}[\theta \odot x^T > 0]$$

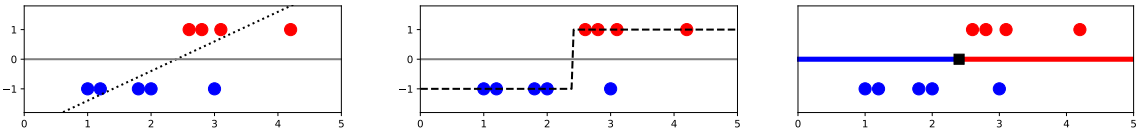
which would instead output zero or one, but be otherwise equivalent.

Linear classifiers are sometimes called "perceptrons", their historical name when originally proposed by Rosenblatt **?** and studied in the early days of artificial intelligence **?**.

4.1 Characterizing a linear classifier

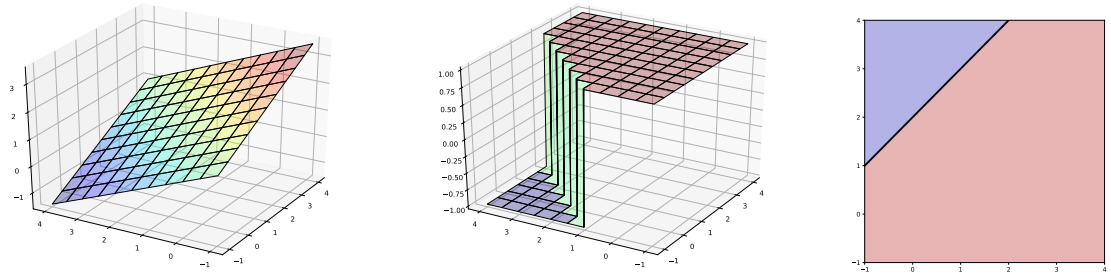
The decision boundary for a linear classifier is also linear. Recall that the decision boundary are the points at which we transition from decision +1 to decision -1; in our learner (4.1), this occurs at precisely the solution to the linear equation $\theta \odot x^T = 0$. For n features, the set of solutions to this equation will be a linear subspace of dimension $n - 1$; so, for example, with two features x_1, x_2 we can solve to find the decision boundary, which is a line in the two-dimensional feature space:

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0 \qquad \Rightarrow \qquad x_2 = -\frac{\theta_1}{\theta_2} x_1 - \frac{\theta_0}{\theta_2}$$



(a) (b) (c)

Figure 4.1: Data in one dimension (a single feature x_1), with vertical axis and color indicating y, along with (a) the linear response $r(x) = \theta_0 + \theta_1 x_1$ and (b) the decision function $\text{sign}(r(x))$. (c) The resulting decision regions have a boundary defined by the point, $r(x) = 0$ (black square).



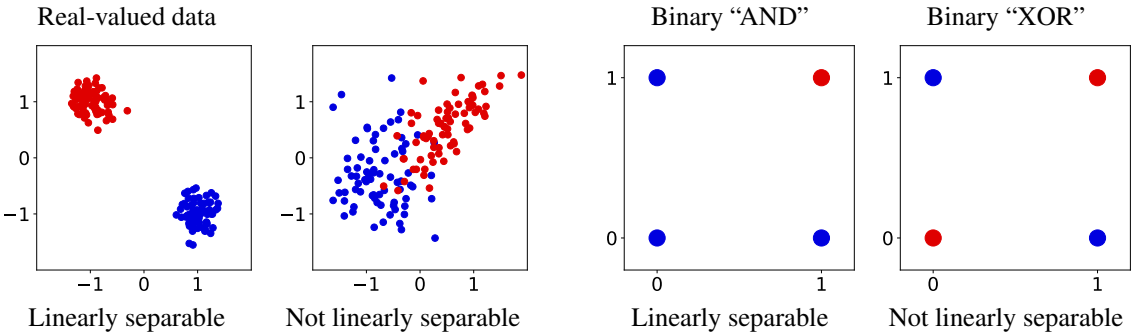
(a) (b) (c)

Figure 4.2: (a) For two-dimensional features (x_1, x_2), the linear response $r(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$, is a 3D linear surface; (b) thresholding produces the decision function $\text{sign}(r(x))$; and (c) the decision regions have a linear decision boundary defined by $r(x) = 0$.

Linearly separable data

It is useful to differentiate between data sets that can be "linearly separated", i.e., there exists a linear classifier that achieves zero training error, and those that cannot. A few examples are shown here, for both real-valued features and binary-valued features.

Example 4-1 : Linear separability



For linearly separable data, there are often many lines that achieve zero error; for example, on the leftmost data, $\text{sign}(x_2)$, $-\text{sign}(x_1)$, and $\text{sign}(x_2 - x_1)$ are three of the many decision functions that separate the two classes. In the two examples on the right, a perceptron (linear classifier) can be used to represent a binary AND function (for example, as $f(x) = \text{sign}(x_1 + x_2 - 1.5)$), but cannot correctly learn an exclusive or (XOR) function, a result famously discussed by Minsky and Papert [1969].

4.2 Training a linear classifier

What makes a good classifier? The usual measure of error for classification is the classification error, or misclassification rate – the number of mistakes (misclassifications) made on the data. Typically, we would like to minimize the misclassification rate over the training data,

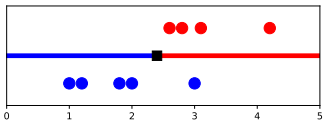
$$J_{01}(\theta) = \frac{1}{m} \sum_i \mathbb{1}[y^{(i)} \neq \text{sign}(x^{(i)} \odot \theta^T)],$$

by finding a set of parameters (or weights) θ that make few errors. As with linear regression, we can notionally think about exploring the space of parameters, assigning each point a cost, and searching for the point with minimum cost.

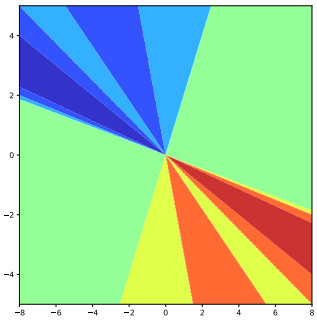
However, the misclassification rate is often difficult to optimize directly. First, it is not smooth – it changes value only when the decision boundary passes a data point, so it is constant until it changes abruptly one way or the other. There is thus very little to signal which direction we should modify the parameters, in order to reduce the error. Another consequence of such "flat" values can be seen when the data are linearly separable – there may be a large set of linear classifiers that achieve zero error, but intuitively we can guess that some are likely to be better than others. Classification accuracy alone, however, judges these classifiers exactly the same way.

Example 4-2 : Misclassification Loss, J_{01}

Consider the same data from Figure 4.1. Evaluating the misclassification loss J_{01} at each possible value of θ, we see that the loss is locally constant and changes discontinuously, making J_{01} difficult to optimize. (The radial appearance arises from the fact that the 1D decision boundary is determined by the value of θ_0/θ_1.)



Data $\{(x^{(i)}, y^{(i)})\}$



Loss $J_{01}(\theta_0, \theta_1)$

Such difficulties motivate the use of "surrogate" loss functions – error functions that we can use to replace the classification error that will be easier for us to optimize. The typical training approach is to learn parameters to optimize the surrogate loss, and hope (sometimes with good reason) that this also produces good classification accuracy.

Linear classification as a regression problem

The linear classifier has the form of thresholding a linear function of the features. How can we learn a good linear function? One simple way we could consider is just to learn a linear predictor of the class by treating the target values $y^{(i)}$ as if they were real-valued, fitting a standard linear regression model, and

then thresholding the predictions to return one of the two discrete classes. In effect, this means that we are fitting our model to minimize the linear regression MSE loss,

$$J_{\text{MSE}}(\theta) = \frac{1}{m}\sum_i \left(y^{(i)} - (x^{(i)} \odot \theta^T)\right)^2$$

rather than the misclassification loss that we are really interested in,

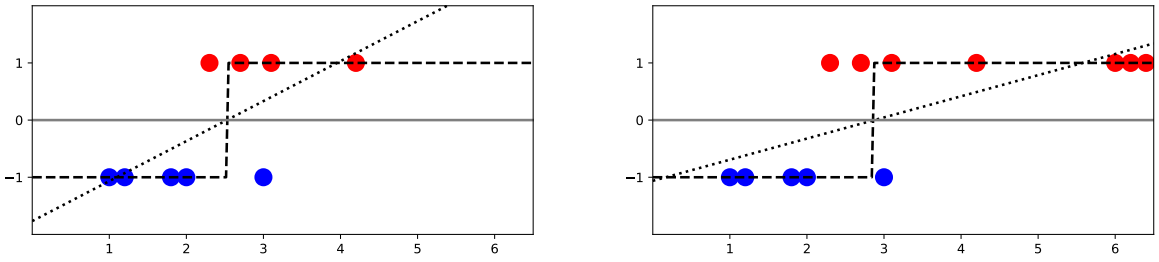
$$J_{01}(\theta) = \frac{1}{4m}\sum_i \left(y^{(i)} - \text{sign}(x^{(i)} \odot \theta^T)\right)^2$$

(where we have used an equivalent form of J_{01} for $y^{(i)} \in \{-1, 1\}$, to emphasize its similarity to J_{MSE}). These losses look similar, and the parameters we find will have the best possible J_{MSE}, but does this mean it will also have a good J_{01}? Unfortunately, the differences between J_{01} and J_{MSE} will often lead to this approach producing a poor classifier. Consider an example:

Example 4-3 : MSE Surrogate Loss

Suppose we have the same data as in Figure 4.1, consisting of five negative and four positive examples, and displayed as a regression problem (with the class Y shown using the vertical axis). Fitting θ using linear regression and the MSE loss produces the dotted line fit in Figure 4.3(a); when this linear approximation to Y is converted to a binary class by the threshold function $T(\cdot)$ (dashed line), it has a reasonable J_{01} loss as well: $J_{01} = 2/9$ (although better settings of θ can acheive $J_{01} = 1/9$ on these data).

However, suppose that we now add three new positive examples, near $X = 6$, as shown in Figure 4.3(b). The previous setting of θ would classify all these points correctly, again giving only two misclassifications. But counter-intuitively, re-optimizing J_{MSE} gives a new value of θ (dotted line) whose thresholded value (dashed line) actually classifies the training data less well! In essence, this behavior is due to the differences between the two losses – the first setting of θ predicts the sign of Y correctly on the three new points, since $x \odot \theta^T$ is large and positive. However, since $x \odot \theta^T$ is large, it is not a very good fit to the data points' *value*, +1. Optimizing J_{MSE} alters θ to be a better fit for the values of Y, leading to more data that have an incorrect sign.



(a) (b)

Figure 4.3: Fitting a classifier using linear regression. (a) Fitting a line to the data points via linear regression, and then thresholding the resulting response, produces a reasonable if sub-optimal classifier with $J_{01} = 2/7$. (b) Adding new, correctly classified data can shift the optimal linear regression fit to classify *less well*, shifting the line to make its linear response value closer to the class values but making more classification mistakes after thresholding.

The perceptron algorithm

If we tweak our linear regression approach slightly, we can arrive at a classical and historically significant algorithm for training linear classifiers, called the *perceptron algorithm*. The perceptron algorithm, described in Algorithm 4.1, looks almost identical to a stochastic gradient descent optimization for minimum squared error linear regression, except for one detail – we use $\hat{y} = \text{sign}(x \odot \theta^T)$ in the update equation, instead of the linear response $x \odot \theta^T$.

This small tweak works to circumvent the problem of "well classified" data points. For any data point i, if our current *class prediction* $\hat{y}^{(i)}$ is correct, then $(y^{(i)} - \hat{y}^{(i)}) = 0$ and θ will not be modified. On the other hand, suppose that $y^{(i)} = +1$, but $\hat{y}^{(i)} = -1$. This means our linear response $x^{(i)} \odot \theta^T$ is negative, and we would like to increase it. So, θ is updated to $\theta + 2\alpha x^{(i)}$, and the linear response for this point will increase by $2\alpha\|x^{(i)}\|^2$.

Algorithm 4.1 Perceptron Algorithm

Initialize parameters θ, step α
repeat
  for $i \in 1 \ldots m$ **do**
    $\hat{y}^{(i)} = \text{sign}(x^{(i)} \odot \theta^T)$
    $\theta \leftarrow \theta + \alpha(y^{(i)} - \hat{y}^{(i)})x^{(i)}$
  end for
until converged

It is easy to see that the fixed points of the perceptron algorithm are linear separators of the data – should we ever find a value of θ that classifies all the training data correctly, all future update steps will leave θ unchanged, and we can declare convergence and stop the algorithm. By analyzing the behavior of this algorithm, one can prove that for any step size $0 < \alpha < 1$, if the data are linearly separable (so that there *exists* a value of θ with zero classification error), the perceptron algorithm will converge to some such fixed point [Novikoff, 1962].

While these results seem promising, they are a bit limited. For one thing, if there are many linear separators, there is no criterion to select *which* of these separators we prefer; the perceptron algorithm simply returns the first such separator that it finds. Secondly, when the data are *not* separable, the algorithm will be non-convergent, and it is unclear when to stop or what solution to return[1]. Finally, since the perceptron algorithm is specified procedurally, it is unclear how to modify it to improve its behavior, for example incorporating more advanced optimization techniques (e.g., Newton's method) or other desired properties of the learner (e.g., regularization). For these reasons, a more popular approach for training linear classifiers is to use a surrogate loss function.

4.3 Surrogate loss functions

A *surrogate* loss function is a loss function $J(\cdot)$ which is used during training as a replacement for the loss in which the user is actually interested. For example, although we are often interested in minimizing our learner's error rate, $J_{01}(\theta)$, this is not an easy loss to optimize. Instead, we can replace J_{01} with a different loss function J' that is easier to use – for example, one that is smooth and differentiable, allowing us to use gradient methods, or even convex, ensuring that there are no local optima. Then, we can use a host of optimization tools and techniques to search for the minimum of J'. Ideally, our surrogate loss should also be similar to the desired loss, in the sense that we are hoping that finding the minimum of J' will be close to the minimum of J. In the sequel, we examine two examples of such surrogates and compare them.

[1] One possible option for the latter is to keep track of the parameters resulting in the best training error rate over the course of the iterations [Gallant, 1990].

Logistic MSE loss

When we attempted to fit the paramters of our perceptron using linear regression, the issue that we observed was that the linear response, $x \odot \theta^T$, was too dissimilar to the model's prediction value, $\text{sign}(x \odot \theta^T)$. So, one possibility is to use a *nonlinear* regression model that is closer to the perceptron output, but still smooth and differentiable.

There are many possibilities, but here we will use one based on the well-known *logistic function* $\sigma(z)$. The logisitic function is a saturating, "sigmoid" function, whose derivative is conveniently expressable in terms of itself:

$$\sigma(z) = 1/(1 + \exp(-z))$$
$$\partial\sigma(z) = \sigma(z) \cdot (1 - \sigma(z))$$

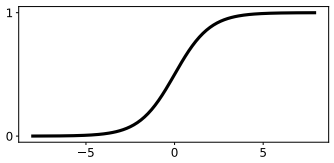


Figure 4.4: Logistic function

The logistic $\sigma(z)$ ranges between zero and one, making it a possible "smooth surrogate" for the indicator function $\mathbb{1}[z > 0]$ (useful if our classes are $y \in \{0, 1\}$; a simple scaling transform, $\rho(z) = 2\sigma(z) - 1$, gives a version that can substitute for $\text{sign}(z)$, in which case $\partial\rho(z) = 2\partial\sigma(z)$.)

Then, we can use an mean squared error loss, but with the (smooth) nonlinearity providing a "soft" prediction which we substitute for $\hat{y}$:

$$J_{\text{LMSE}}(\theta) = \frac{1}{m} \sum_i \left(y^{(i)} - \rho(x^{(i)} \odot \theta^T) \right)^2 \qquad \text{for } y^{(i)} \in \{-1, +1\}.$$

Since J_{LMSE} is smooth, we can use gradient descent to find the parameters θ; it is easy to derive that the gradient is,

$$\nabla J^{(i)}_{\text{LMSE}}(\theta) = -(y^{(i)} - \rho(x^{(i)} \odot \theta^T)\, \partial\rho(x^{(i)} \odot \theta^T)\, x^{(i)}$$

and we can optimize θ via gradient descent or any of its many variants. Note that J_{LMSE} is used only for the training process – our prediction should still be one of the two class values, and so we continue to use the predictor (4.1).

Logistic Regression

Another common surrogate loss for linear classifiers is also based on the logistic function, but interprets its output as a class probability, rather than a real-valued prediction. From this perspective, take $\sigma(x \odot \theta^T)$ to be the probability that the point has class $+1$. Then, the average negative log-likelihood of our training observations is,

$$J_{\text{NLL}}(\theta) = \frac{1}{m} \sum_i J^{(i)}(\theta) \qquad J^{(i)}(\theta) = \begin{cases} -\log[\sigma(x^{(i)} \odot \theta^T)] & \text{if } y^{(i)} = +1 \\ -\log[1 - \sigma(x^{(i)} \odot \theta^T)] & \text{if } y^{(i)} = -1 \end{cases}$$

This particular model and loss function goes by the historical name *logistic regression*. To see why, suppose that we had access to the true conditional probability $p(Y|X)$, and let $r(x)$ to be the log-odds ratio of the two classes; then we can see that

$$r(x) = \log \left[\frac{p(Y = +1|X = x)}{p(Y = -1|X = x)} \right] \qquad \Leftrightarrow \qquad p(Y = +1|X = x) = \sigma(r(x)),$$

i.e., the logistic function transforms the log-odds ratio r to the probability of the positive class. So, fitting a probability model with the assumption that $p(Y = +1|X = x) \approx \sigma(x \odot \theta^T)$ can be viewed as fitting a linear approximation to the log-odds ratio, $r(x) \approx x \odot \theta^T$.

The loss J_{NLL} is also smooth and differentiable, with derivative

$$\nabla J_{\text{NLL}}(\theta) = \frac{1}{m}\sum_i \nabla J^{(i)}_{\text{NLL}}(\theta) \qquad \nabla J^{(i)}_{\text{NLL}} = \left(\sigma(x^{(i)} \odot \theta^T) - \mathbb{1}[y^{(i)} = +1]\right) x^{(i)}$$

A nice property of the logistic negative log-likelihood loss is that it is *convex*, and thus has no local minima (any local minimum is also a global minimum). For convex functions, even local search methods like gradient descent will eventually converge to the optimum, making model initialization less critical. Many optimization techniques can also take advantage of convexity to speed up learning. To see that $J_{\text{NLL}}(\theta)$ is convex, simply take the second derivative,

$$\frac{\partial^2}{\partial\theta_j\,\partial\theta_k} J_{\text{NLL}}(\theta) = \left[\nabla^2 J_{\text{NLL}}(\theta)\right]_{jk} = \frac{1}{m}\sum_i \left(\sigma(x^{(i)} \odot \theta^T)(1 - \sigma(x^{(i)} \odot \theta^T))\right) x^{(i)}_j\, x^{(i)}_k$$

and notice that the matrix of second derivatives $\nabla^2 J$ (the Hessian of J) is a weighted empirical covariance of the data, where the weight $\sigma(1-\sigma)$ is close to zero when the model predicts either class with high confidence, and at its maximum ($= \frac{1}{4}$) when the predicted probability is $\sigma(\cdot) = \frac{1}{2}$. Thus, $\nabla^2 J$ is symmetric and positive semi-definite, which is sufficient to show that $J_{\text{NLL}}(\theta)$ is convex.

Properties of surrogate losses

There are many possible surrogate losses; here we have seen two, and in later chapters we discuss several more. Both these surrogate losses, J_{LMSE} and J_{NLL}, are smooth and differentiable, enabling them to be trained using a variety of techniques.

One property that many good surrogate functions possess is that they can be used to bound the desired loss function. For example, both J_{LMSE} and J_{NLL} bound J_{01} up to a multiplicative constant: it is easy to show for $y \in \{-1, +1\}$ that for all θ,

$$J_{01}(\theta) \le J_{\text{LMSE}}(\theta) \qquad J_{01}(\theta) \le \frac{1}{\log 2} J_{\text{NLL}}(\theta).$$

simply by evaluating the losses at $x \odot \theta^T = 0$. Such a bounding property is useful – if we do manage to optimize the surrogate to a very low cost value, it tells us that the loss we actually care about cannot be too much higher.

We can also visualize some of the differences between these losses by examining how they behave as a function of the "distance" from the decision boundary, i.e., as a function of $x \odot \theta^T$. In Figure 4.5, take $y = +1$; then J_{01} transitions abruptly from no cost to unit cost when $x \odot \theta^T \le 0$. J_{LMSE} transitions smoothly, and as our linear response becomes more and more negative (confident in the wrong answer), saturates to a constant cost ($J = 4$). Finally, J_{NLL} is smooth and continues to increase in cost as our response becomes more negative.

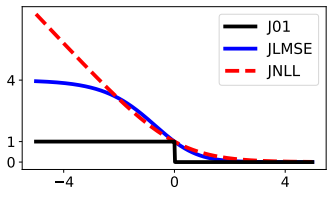


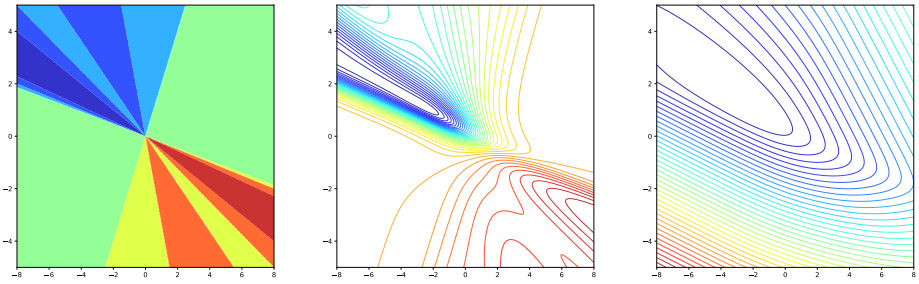
Figure 4.5: Surrogate losses

The shape of our surrogate loss can have a number of effects on the classifier we end up learning. For example, the J_{LMSE} loss, which saturates at maximum value 4 far from the decision boundary, might be described as "robust to outliers" – a small number of "bad" data, no matter how far away from the decision boundary, will have a limited ability to change the decision boundary's location (since they can only increase the total cost by a small amount). On the other hand, the further away these data are, the more they can increase the cost of loss J_{NLL}, so that even a small number of sufficiently distant data can distort the optimum of J_{NLL}. On the other hand, however, the same saturation property means that the

gradient $J^{(i)}_{\text{LMSE}}$ of data i that are far from the boundary is very small; this can make optimization very slow and prone to local optima, especially if the model is initialized poorly. Meanwhile, the convexity of J_{NLL} means that we can always find the same (global) optimum, independent of initialization.

Example 4-4 : Comparing surrogate losses (1)

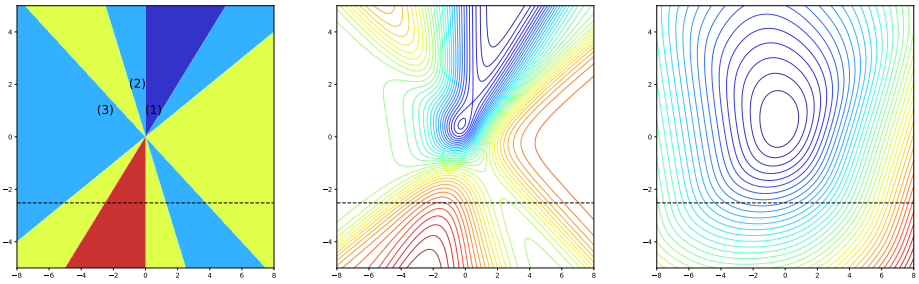
If we look at the surrogate loss functions on the same 1D data from Figure 4.1, we find that both surrogates, J_{LMSE} and J_{NLL}, are reasonably similar to the misclassification loss, J_{01}. In this case, both surrogates are reasonably well-behaved and easy to optimize, and visually similar to the original J_{01}.



$J_{01}(\theta)$ $J_{\text{LMSE}}(\theta)$ $J_{\text{NLL}}(\theta)$

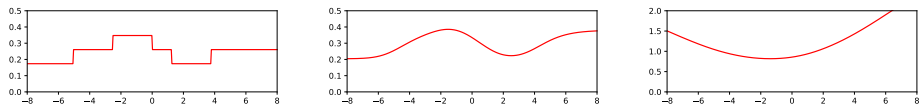
Example 4-5 : Comparing surrogate losses (2)

Taking a different set of data (**show data?**), we can see more differences between the surrogates. In this case, J_{LMSE} is more visually similar to J_{01} than J_{NLL}:



$J_{01}(\theta)$ $J_{\text{LMSE}}(\theta)$ $J_{\text{NLL}}(\theta)$

However, this comes at the cost of being non-convex and harder to optimize. To see this, we show the value of J for a fixed θ_1, while varying θ_0 (i.e., the dashed line). Doing so, we observe that J_{LMSE} looks essentially like a smoothed version of the up-and-down loss of J_{01}, while J_{NLL} instead gives a convex function with a unique minimum:



$J_{01}(\theta_0, \theta_1 = V)$ $J_{\text{LMSE}}(\theta_0, \theta_1 = V)$ $J_{\text{NLL}}(\theta_0, \theta_1 = V)$

4.4 Feature transforms and complexity

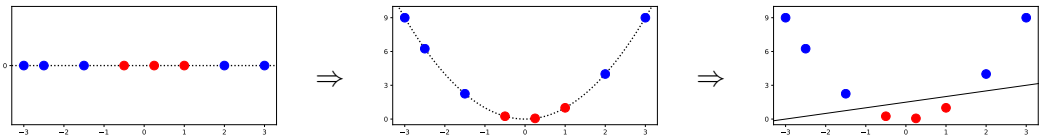
Feature expansions, like polynomial functions, effect on r(x) and T(r(x)) Linear response in expanded dimension (data live on non-linear sub-manifold) vs nonlinear response in original feature space

Example 4-6 : Feature transforms: polynomial

Suppose we are given eight data points $(x^{(i)}, y^{(i)})$:

$$D = \{(-3,-1),(-2.5,-1),(-1.5,-1),\ (-0.5,+1),(0.25,+1),(1,+1),\ (2,-1),(3,-1)\}.$$

These data ar not linearly separable (since the positive class examples lie in between two sets of negative examples). However, if we expand our feature set to create a new feature, $\Phi(x) = [x, x^2] = [\tilde{x}_1, \tilde{x}_2]$, we can visualize our data points in a two-dimensional feature space. Since $\tilde{x}_2$ is a deterministic function of $\tilde{x}_1$, the data will all lie on a curve, $\tilde{x}_2 = \tilde{x}_1^2$. But, in this new two-dimensional space, the positive data are separable from the negative data:



Data $\{(x, y)\}$ $\{(\tilde{x}_1, \tilde{x}_2, y)\}$ Decision boundary

The indicated decision boundary corresponds to the decision function, $T(-2\tilde{x}_2 + \tilde{x}_1 + 3)$, whose response function is quadratic in the original (scalar) feature x, but linear in the new, two-dimensional features $\tilde{x}$.

4.5 Multi-class linear classifiers

Up to this point, we have focused on linear classifiers for binary-valued targets, e.g., $Y \in \{-1, 1\}$. However, it is relatively straightforward to extend all the same techniques to multi-class settings.

Suppose that $Y \in \{1, \ldots, C\}$. Instead of a single parameter vector θ, let us associate a vector θ_c for each class $1 \ldots C$. Then, we define our decision function, $f(x)$, to be the class with the largest linear response:

$$\hat{y} = f(x\,;\,\{\theta_c\}) = \arg\max_c\ \theta_c \odot x^T$$

(or more generally $\theta_c \odot \Phi(x)^T$ for any desired feature transform Φ). In a practical sense, we can arrange the θ_c in a $C \times n$ matrix Θ, to compute the C linear responses $r_c(x) = \theta_c \odot x^T$ as a matrix-vector product, $\Theta \odot x^T$, then simply predict whichever row contains the maximum.

Decision boundary. The decision boundary for a multi-class linear classifier is piecewise linear. Consider two of the class values, c and c'. We prefer c to c' if,

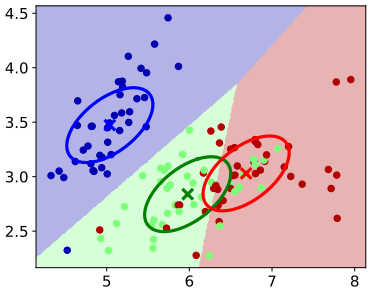
$$\theta_c \odot x^T > \theta_{c'} \odot x^T \qquad \Leftrightarrow \qquad (\theta_c - \theta_{c'}) \odot x^T > 0,$$

i.e., the decision boundary between c and c' is a linear hyperplane, equivalent to a perceptron defined by $\theta_c - \theta_{c'}$. The overall decision boundary for $f(x; \{\theta_c\})$ is thus a subset of these hyperplanes; whenever

c and c' are the top two linear responses, their hyperplane defines a boundary at which our decision switches from c to c' or vice-versa. (Conversely of course, if in the vicinity of some point x, a different class c'' has $r_{c''}(x)$ larger than r_c and $r_{c'}$, then which of c or c' is preferred is irrelevant; neither c nor c' will be our prediction.)

Example 4-7 : Iris data perceptron

Update; fill in.



Alternative form. It is sometimes convenient to re-define the multi-class linear model in a slightly more general way, by

$$\hat{y} = f(x\,;\,\theta) = \arg\max_c\ \theta \odot \Phi(x,c)^T$$

in which the class values c are incorporated into the feature transform Φ. This formulation generalizes our previous version, since we can always choose,

$$\begin{aligned} \theta &= \big[\quad \theta_1 \quad , \quad \theta_2 \quad , \quad \dots \big] \\ \Phi(x,c) &= \big[x\,\mathbb{1}[c=1]\ ,\ x\,\mathbb{1}[c=2]\ ,\quad \dots\ \big] \end{aligned}$$

so that θ and $\Phi(x,c)$ are vectors of size $1 \times (nC)$, and $\Phi(x,c)$ is zero everywhere except in the n entries corresponding to θ_c.

This formulation is useful in some structured settings. For example, if we have certain classes that are related or in a category (say, two types of dog), we can allow them to share parameters by having those features be non-zero for several classes c (e.g., when c is any type of dog). Similarly, we can allow different classes c to have different numbers of features, for example when some features only make sense or are well-defined for certain classes c.

Multi-class perceptron algorithm. We can easily generalize the perceptron algorithm (4.1) to the multi-class case. After selecting our prediction, $\hat{y}^{(i)} = \arg\max_c\ \theta \odot \Phi(x,c)$, we simply increase the response of the correct y, and decrease that of $\hat{y}$:

$$\theta \leftarrow \theta + \alpha\big(\Phi(x^{(i)}, y^{(i)}) - \Phi(x^{(i)}, \hat{y}^{(i)})\big).$$

As before, any data point i that is correctly predicted will result in no update to θ, making any linear separator of the data a fixed point of the algorithm. However, the drawbacks of the perceptron algorithm (non-convergence, etc.) also carry over to the multi-class case, so that in practice it is more common to use a surrogate loss instead.

Surrogate losses. The same surrogate losses used for the binary classification problem can be easily extended to the multiclass case. To do so, let us first re-frame the prediction of a class $y \in \{1, \ldots, C\}$ as equivalent to the prediction of a length-C "one-hot" vector, $\mathbf{y} \in \{[1, 0, 0, \ldots], [0, 1, 0, \ldots], \ldots\}$, i.e., the vector $\mathbf{y}$ is all zeros except for a single one at position y.

Now, we can use the linear responses r_c to define a prediction of $\mathbf{y}$. For example, we can use the *multi-logistic transform* to generalize the logistic transform used in the binary case:

$$\hat{\mathbf{y}}_c = \frac{\exp(r_c(x))}{\sum_{c'} \exp(r_{c'}(x))}$$

producing a length-C vector $\hat{\mathbf{y}}$ with $\sum_c \hat{\mathbf{y}}_c = 1$. This function is sometimes called the **softmax** function: if one class (say, class 0) has much larger response than the other classes, $r_0(x) \gg r_c(x)$ for $c > 0$, we can see that $\hat{\mathbf{y}} \approx [1, 0, 0, \ldots]$, so that $\hat{\mathbf{y}}$ is close to a one-hot encoding of the maximizing class value.

Then, the LMSE surrogate loss becomes,

$$J_{\text{LMSE}}(\theta) = \frac{1}{m} \sum_i \|\mathbf{y}^{(i)} - \hat{\mathbf{y}}(x^{(i)})\|^2$$

and, intepreting the entries of $\hat{\mathbf{y}}$ as probabilities, we can define the negative log-likelihood loss,

$$J_{\text{NLL}}(\theta) = -\frac{1}{m} \sum_i \log \left(\mathbf{y}^{(i)} \odot \hat{\mathbf{y}}(x^{(i)})^T\right);$$

since $\mathbf{y}$ has only one non-zero entry, the dot product extracts the predicted probability $\hat{\mathbf{y}}_y$ of true class y. This loss is also sometimes called the *cross-entropy* loss.

Example soft-max for Iris data?