CHAPTER

5 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

$$\hat{y} = f(x\ \theta) = \text{sign}(\theta \cdot x^T) = \begin{cases} +1 & \theta \cdot x^T > 0 \\ -1 & \text{otherwise} \end{cases} \tag{5.1}$$

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 $r = x \cdot \theta^T$; if we were to instead select our binary classes to be $\{1, 0\}$, we could alternatively define

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

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 **?**; they are a very simple type of neural network.

5.1 Characterizing a linear classifier

The decision boundary for a linear classifier is also linear. The decision boundary are the points at which we transition from decision +1 to decision -1; in our learner (5.1), this occurs at precisely the solution to the linear equation $\theta \cdot x^T = 0$. 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:

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

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:

As can be seen, a perceptron / linear classifier cannot correctly learn an exclusive or (XOR) function, a result famously discussed by Minsky and Papert in 1969 **?**.

5.2 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, by finding a set of parameters (weights) 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 "signal" about 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.

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 in the "regression" sense, and then threshold that value. So, for example, we define a variable $y^{(i)} = c^{(i)}$, equal to the class value c, to predict using regression. We then regress the data $\{(x^{(i)}, y^{(i)})\}$ to find parameters θ, and finally, we define our (real-valued) regression prediction as $\hat{y}(x) = x \cdot \theta^T$ and our "discrete" class prediction as $\hat{c}(x) = T(\hat{y}(x))$.

This is an example of a surrogate loss. We are normally interested in our misclassification rate, i.e., the fraction of data on which $\hat{c}(x^{(i)}; \theta) \neq c^{(i}$, or:

$$J_{01}(\theta) = \frac{1}{M} \sum_i \mathbb{1}[c^i \neq T(x^{(i)} \cdot \theta^T)]$$

But, our training of the model parameters θ using linear regression actually minimizes a different cost, the MSE of a linear predictor:

$$J_{MSE}(\theta) = \frac{1}{M} \sum_i (\, c^i - (x^{(i)} \cdot \theta^T)\,)^2$$

So the model we find will have a good J_{MSE}, but will it also have a good J_{01}?

Often, it will not. **details; example, etc.**

The perceptron algorithm

Surrogate loss functions

5.3 Logistic Regression

Logistic regression is used for predicting values between zero and one – for example, probabilities. For this reason it is commonly used as a binary classification method, where the two class values are taken to be zero or one.

The logistic function has a "sigmoid" shaped response, i.e., it looks like a flattened-out "S". Its functional form, and its derivative, are given by

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

UPDATE

We use the logistic function as a soft thresholding operation, to replace the perceptron's hard threshold T(). We maintain the linear "internal" form, i.e., given a feature vector x and parameters θ, often called the "weights" of the learner, our regression prediction is:

$$\hat{y} = f(x; \theta) = \sigma(x \cdot \theta^T)$$

This gives a real-valued prediction between zero and one.

```
def logistic_response(theta, X)
# Evaluate a logistic regression function with parameters theta
  z = theta[0] + X.dot(theta[1:]);    # Compute linear response
  return 1 / (1 + np.exp(-z));        # apply logistic sigmoid f'n
```

In classification, of course, we need to predict a "discrete" class value; just as with the perceptron, we can simply threshold our real-valued predictor:

$$\hat{y} = T(\hat{y}) = \begin{cases} 0 & \hat{y} < 0.5 \\ 1 & \hat{y} \geq 0.5 \end{cases}$$

Because the logistic is smooth, we will be able to train it using gradient descent.

Logistic MSE Loss

In your homework, you are asked to perform online gradient descent. In "online" gradient descent, we define a cost function relative to a single data point i, for example the squared error of that data point alone: $C_i = (y^{(i)} - \sigma(x^{(i)})^2$. As in standard gradient descent, we calculate the derivative of C_i:

$$\nabla C_i(x_i, y, \theta) = -(y - \sigma(x^{(i)}\theta^T) \cdot \partial\sigma(x^{(i)}\theta^T) \cdot x^{(i)}$$

and use this gradient to make an update to the parameter vector θ, using a step-size parameter α:

$$\theta \leftarrow \theta - \alpha\nabla C_i$$

In Python, this is:

```
X1 = np.hstack(( np.ones(M,1), X ))      # prepend x0, the constant feature
for i in np.random.permutation(M):       # for each data point
   s = logistic_response(theta,X[i,:]);  # compute the ith prediction
   grad = (s - Y[i]) * s*(1-s) * X1[i,:];
   theta = theta - stepsize * grad;      # take a step down the gradient

S = logistic_response(wts, X);          # compute all outputs,
mse[iter] = ( (Y-S)**2 ).mean();        #   the overall MSE
err[iter] = ( Y != (S>0.5) ).mean();    #   and the error rate
```