Lecture 3, Part 1: Linear Classification

Roger Grosse

1 Introduction

Last time, we saw an example of a learning task called regression. There, the goal was to predict a scalar-valued target from a set of features. This time, we'll focus on a slightly different task: **binary classification**, where the goal is to predict a *binary-valued* target. Here are some examples of binary classification problems:

- You want to train a medical diagnosis system to predict whether a patient has a given disease. You have a training set consisting of a set of patients, a set of features for those individuals (e.g. presence or absence of various symptoms), and a label saying whether or not the patient had the disease.
- You are running an e-mail service, and want to determine whether a given e-mail is spam. You have a large collection of e-mails which have been hand-labeled as spam or non-spam.
- You are running an online payment service, and want to determine whether or not a given transaction is fraudulent. You have a labeled training dataset of fraudulent and non-fraudulent transactions; features might include the type of transaction, the amount of money, or the time of day.

Like regression, binary classification is a very restricted kind of task. Most learning problems you'll encounter won't fit nicely into one of these two categories. Our motivation for focusing on binary classification is to introduce several fundamental ideas that we'll use throughout the course. In this lecture, we discuss how to view both data points and linear classifiers as vectors. Next lecture, we discuss the perceptron, a particular classification algorithm, and use it as an example of how to efficiently implement a learning algorithm in Python. Starting next week, we'll look at supervised learning in full generality, and see that regression and binary classification are just special cases of a more general learning framework.

This lecture focuses on the geometry of classification. We'll look in particular at two spaces:

- The input space, where each data case corresponds to a vector. A classifier corresponds to a decision boundary, or a hyperplane such that the positive examples lie on one side, and negative examples lie on the other side.

- Weight space, where each set of classification weights corresponds to a vector. Each training case corresponds to a constraint in this space, where some regions of weight space are "good" (classify it correctly) and some regions are "bad" (classify it incorrectly).

The idea of weight space may seem pretty abstract, but it is very important that you become comfortable with it, since it underlies nearly everything we do in the course.

Using our understanding of input space and weight space, the limitations of linear classifiers will become immediately apparent. We'll see some examples of datasets which are not linearly separable (i.e. no linear classifier can correctly classify all the training cases), but which become linearly separable if we use a basis function representation.

1.1 Learning goals

- Know what is meant by binary linear classification.
- Understand why an explicit threshold for a classifier is redundant. Understand how we can get rid of the bias term by adding a "dummy" feature.
- Be able to specify weights and biases by hand to represent simple functions (*e.g.* AND, OR, NOT).
- Be familiar with input space and weight space.
  - Be able to plot training cases and classification weights in both input space and weight space.
- Be aware of the limitations of linear classifiers.
  - Know what is meant by convexity, and be able to use convexity to show that a given set of training cases is not linearly separable.
  - Understand how we can sometimes still separate the classes using a basis function representation.

2 Binary linear classifiers

We'll be looking at classifiers which are both **binary** (they distinguish between two categories) and **linear** (the classification is done using a linear function of the inputs). As in our discussion of linear regression, we assume each input is given in terms of D scalar values, called **input dimensions** or **features**, which we think summarize the important information for classification. (Some of the features, e.g. presence or absence of a symptom, may in fact be binary valued, but we're going to treat these as real-valued anyway.) The jth feature for the ith training example is denoted $x_j^{(i)}$. All of the features for a given training case are concatenated together to form a vector, which we'll denote $\mathbf{x}^{(i)}$. (Recall that vectors and matrices are shown in boldface.)

Associated with each data case is a binary-valued **target**, the thing we're trying to predict. By definition, a binary target takes two possible values,

which we'll call **classes**, and which are typically referred to as **positive** and **negative**. (E.g., the positive class might be "has disease" and the negative class might be "does not have disease.") Data cases belonging to these classes are called **positive examples** and **negative examples**, respectively. The **training set** consists of a set of N pairs $(\mathbf{x}^{(i)}, t^{(i)})$, where $\mathbf{x}^{(i)}$ is the input and $t^{(i)}$ is the binary-valued target, or **label**. Since the training cases come with labels, they're referred to as **labeled examples**. Confusingly, even though we talk about positive and negative examples, the $t^{(i)}$ typically take values in $\{0, 1\}$, where 0 corresponds to the "negative" class. Sorry, you'll just have to live with this terminology.

Our goal is to correctly classify all the training cases (and, hopefully, examples not in the training set). In order to do the classification, we need to specify a **model**, which determines how the predictions are computed from the inputs. As we said before, our model for this week is binary linear classifiers.

The way binary linear classifiers work is simple: they compute a linear function of the inputs, and determine whether or not the value is larger than some **threshold** r. Recall from Lecture 2 that a linear function of the input can be written as

$$w_1 x_1 + \cdots + w_D x_D + b = \mathbf{w}^T \mathbf{x} + b,$$

where $\mathbf{w}$ is a **weight vector** and b is a scalar-valued **bias**. Therefore, the prediction y can be computed as follows:

$$\begin{aligned} z &= \mathbf{w}^T \mathbf{x} + b \\ y &= \begin{cases} 1 & \text{if } z \geq r \\ 0 & \text{if } z < r \end{cases} \end{aligned}$$

This is the model we'll use for the rest of the week.

2.1 Thresholds and biases

Dealing with thresholds is rather inconvenient, but fortunately we can get rid of them entirely. In particular, observe that

$$\mathbf{w}^T \mathbf{x} + b \geq r \iff \mathbf{w}^T \mathbf{x} + b - r \geq 0.$$

In other words, we can obtain an equivalent model by replacing the bias with $b - r$ and setting r to 0. From now on, we'll assume (without loss of generality) that the threshold is 0. Therefore, we rewrite the model as follows:

$$\begin{aligned} z &= \mathbf{w}^T \mathbf{x} + b \\ y &= \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \end{aligned}$$

In fact, it's possible to eliminate the bias as well. We simply add another input dimension x_0, called a **dummy feature**, which always takes the value 1. Then

$$w_0 x_0 + w_1 x_1 + \cdots + w_D x_D = w_0 + w_1 x_1 + \cdots + w_D x_D,$$

so w_0 effectively plays the role of a bias. We can then simply write

$$z = \mathbf{w}^T \mathbf{x}.$$

Eliminating the bias often simplifies the statements of algorithms, so we'll sometimes use it for notational convenience. However, you should be aware that, when actually implementing the algorithms, the standard practice is to include the bias parameter explicitly.

2.2 Some examples

Let's look at some examples of how to represent simple functions using linear classifiers — specifically, AND, OR, and NOT.

Example 1. *Let's start with NOT, since it only involves a single input. Here's a "training set" of inputs and targets we're trying to match:*

x_1	t
0	*1*
1	*0*

Each of the training cases provides a constraint on the weights and biases. Let's start with the first training case. If $x_1 = 0$, then $t = 1$, so we need $z = w_1 x_1 + b = b \geq 0$. Technically we could satisfy this constraint with $b = 0$, but it's good practice to avoid solutions where z lies on the decision boundary. Therefore, let's tentatively set $b = 1$.

Now let's consider the second training case. The input is $x_1 = 1$ and the target is $t = 0$, so we need $z = w_1 \cdot 1 + b = w_1 + 1 < 0$. We can satisfy this inequality with $w_1 = -2$. This gives us our solution: $w_1 = -2$, $b = 1$.

Example 2. *Now let's consider AND. This is slightly more complicated, since we have 2 inputs and 4 training cases. The training cases are as follows:*

x_1	x_2	t
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

Just like in the previous example, we can start by writing out the inequalities corresponding to each training case. We get:

$$\begin{aligned} b &< 0 \\ w_2 + b &< 0 \\ w_1 + b &< 0 \\ w_1 + w_2 + b &> 0 \end{aligned}$$

From these inequalities, we immediately see that $b < 0$ and $w_1, w_2 > 0$. The simplest way forward at this point is probably trial and error. Since the problem is symmetric with respect to w_1 and w_2, we might as well decide that $w_1 = w_2$. So let's try $b = -1, w_1 = w_2 = 1$ and see if it works. The first and fourth inequalities are clearly satisfied, but the second and third are not, since $w_1 + b = w_2 + b = 0$. So let's try making the bias a bit more negative. When we try $b = -1.5, w_1 = w_2 = 1$, we see that all four inequalities are satisfied, so we have our solution.

Following these examples, you should attempt the OR function on your own.

3 The geometric picture

Now let's move on to the main concepts of this lecture: data space and weight space. These are the spaces that the inputs and the weight vectors live in, respectively. It's very important to become comfortable thinking about these spaces, since we're going to treat the inputs and weights as vectors for the rest of the term.

In this lecture, we're going to focus on two-dimensional input and weight spaces. But keep in mind that this is a vast oversimplification: in practical settings, these spaces are typically many thousands, or even millions, of dimensions. It's pretty much impossible to visualize spaces this high-dimensional.

3.1 Data space

The first space to be familiar with is **data space**, or **input space**. Each point in this space corresponds to a possible input vector. (We're going to abuse mathematical terminology a bit by using "point" and "vector" interchangeably.) It's customary to represent positive and negative examples with the symbols "+" and "−", respectively.

Once we've chosen the weights $\mathbf{w}$ and bias b, we can divide the data space into a region where the points are classified as positive (the **positive region**), and a region where the points are classified as negative (the **negative region**). The boundary between these regions, i.e. the set where $\mathbf{w}^T\mathbf{x} + b = 0$, is called the **decision boundary**. Think back to your linear algebra class, and recall that the set determined by this equation is a **hyperplane**. The set of points on one side of the hyperplane is called a **half-space**. Examples are shown in Figure 1

When we plot examples in two dimensions, the hyperplanes are actually lines. But you shouldn't think of them as lines — you should think of them as hyperplanes.

If it's possible to choose a linear decision boundary that correctly classifies all of the training cases, the training set is said to be **linearly separable**. As we'll see later, not all training sets are linearly separable.

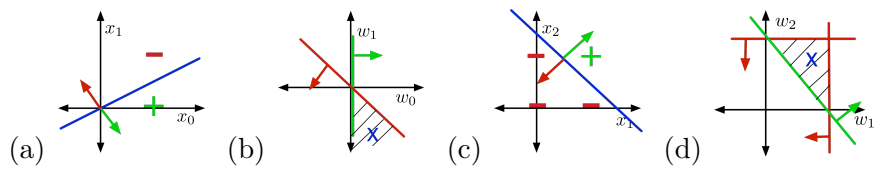

Figure 1: **(a)** Training examples and for NOT function, in data space. **(b)** NOT, in weight space. **(c)** Slice of data space for AND function corresponding to $x_0 = 1$. **(d)** Slice of weight space for AND function corresponding to $w_0 = -1$.

3.2 Weight space

As you'd expect from the name, weight vectors are also vectors, and the space they live in is called **weight space**. In this section, we'll assume there is no bias parameter unless stated otherwise. (See Section 2.1.) Each point in weight space is a possible weight vector.

Consider a positive training case $(\mathbf{x}, 1)$. The set of weight vectors which correctly classify this training case is given by the linear inequality $\mathbf{w}^T\mathbf{x} \geq 0$. (In fact, it's exactly the sort of inequality we derived in Examples 1 and 2.) Geometrically, the set of points satisfying this inequality is a half-space. For lack of a better term, we'll refer to the side which satisfies the constraint as the **good region**, and the other side as the **bad region**. Similarly, the set of weight vectors which correctly classify a negative training case $(\mathbf{x}, 0)$ is given by $\mathbf{w}^T\mathbf{x} < 0$; this is also a half-space. Examples are shown in Figure 1.

We're going to completely ignore the fact that one of these inequalities is strict and the other is not. The question of what happens on the decision boundaries isn't very interesting.

The set of weight vectors which correctly classify *all* of the training examples is the intersection of all the half-spaces corresponding to the individual examples. This set is called the **feasible region**. If the feasible region is nonempty, the problem is said to be **feasible**; otherwise it's said to be **infeasible**.

When we draw the constraints in two dimensions, we typically draw the line corresponding to the boundary of the constraint set, and then indicate the good region with an arrow. As with our data space visualizations, you should think of the boundary as a hyperplane, not as a line.

There's one constraint per training example. What happened to the fourth constraint in Figure 1(d)?

We can visualize three-dimensional examples by looking at slices. As shown in Figure 2, these slices will resemble our previous visualizations, except that the decision boundaries and constraints need not pass through the origin.

4 The perceptron learning rule

The perceptron is a kind of binary linear classifier. Recall from last lecture that this means it makes predictions by computing $\mathbf{w}^T\mathbf{x}+b$ and seeing if the result is positive or negative. Here, $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the weight vector, and b is a scalar-valued bias. Recall as well that we can eliminate the bias by adding a dummy dimension to $\mathbf{x}$. For the perceptron algorithm, it will be convenient to represent the positive and negative classes with 1 and -1, instead of 1 and 0 as we use in the rest of the course. Therefore,

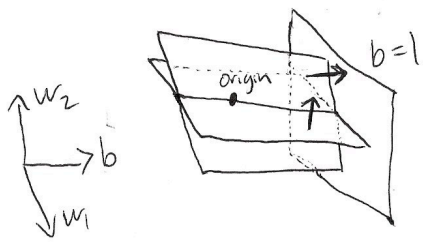

Figure 2: Visualizing a slice of a 3-dimensional weight space.

the classification model is as follows:

$$z = \mathbf{w}^T \mathbf{x} \tag{1}$$

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{2}$$

Here's a rough sketch of the **perceptron algorithm**. We examine each of the training cases one at a time. For each input $\mathbf{x}^{(i)}$, we compute the prediction $y^{(i)}$ and see if it matches the target $t^{(i)}$. If the prediction is correct, we do nothing. If it is wrong, we adjust the weights in a direction that makes it more correct.

Now for the details. First of all, how do we determine if the prediction is correct? We could simply check if $y^{(i)} = t^{(i)}$, but this has a slight problem: if $\mathbf{x}^{(i)}$ lies exactly on the classification boundary, it is technically classified as positive according to the above definition. But we don't want our training cases to lie on the decision boundary, since this means the classification may change if the input is perturbed even slightly. We'd like our classifiers to be more robust than this. Instead, we'll use the stricter criterion

$$z^{(i)} t^{(i)} > 0. \tag{3}$$

You should now check that this criterion correctly handles the various cases that may occur.

The other question is, how do we adjust the weight vector? If the training case is positive and we classify it as negative, we'd like to increase the value of z. In other words, we'd like

$$z' = \mathbf{w}'^T \mathbf{x} > \mathbf{w}^T \mathbf{x} = z, \tag{4}$$

where $\mathbf{w}'$ and $\mathbf{w}$ are the new and old weight vectors, respectively. The perceptron algorithm achieves this using the update

$$\mathbf{w}' = \mathbf{w} + \alpha \mathbf{x}, \tag{5}$$

where $\alpha > 0$. We now check that (4) is satisfied:

$$\mathbf{w}'^T \mathbf{x} = (\mathbf{w} + \alpha \mathbf{x})^T \mathbf{x} \tag{6}$$

$$= \mathbf{w}^T \mathbf{x} + \alpha \mathbf{x}^T \mathbf{x} \tag{7}$$

$$= \mathbf{w}^T \mathbf{x} + \alpha \|\mathbf{x}\|^2. \tag{8}$$

Here, $\|\mathbf{x}\|$ represents the Euclidean norm of $\mathbf{x}$. Since the squared norm is always positive, we have $z' > z$.

Conversely, if it's a negative example which we mistakenly classified as positive, we want to decrease z, so we use a negative value of α. Since it's possible to show that the absolute value of α doesn't matter, we generally use $\alpha = 1$ for positive cases and $\alpha = -1$ for negative cases. We can denote this compactly with

$$\mathbf{w} \leftarrow \mathbf{w} + t\mathbf{x}. \tag{9}$$

This rule is known as the **perceptron learning rule**.

Now we write out the perceptron algorithm in full:

For each training case $(\mathbf{x}^{(i)}, t^{(i)})$,

$z^{(i)} \leftarrow \mathbf{w}^T \mathbf{x}^{(i)}$
If $z^{(i)} t^{(i)} \leq 0$,
$\quad \mathbf{w} \leftarrow \mathbf{w} + t^{(i)} \mathbf{x}^{(i)}$

In thinking about this algorithm, remember that we're denoting the classes with -1 and 1 (rather than 0 and 1, as we do in the rest of the course).

5 The limits of linear classifiers

Linear classifiers can represent a lot of things, but they can't represent everything. The classic example of what they can't represent is the XOR function. It should be pretty obvious from inspection that you can't draw a line separating the two classes. But how do we actually prove this?

5.1 Convex sets

An important geometric concept which helps us out here is **convexity**. A set S is convex if the line segment connecting any two points in S must lie within S. It's not too hard to show that if S is convex, then any **weighted average** of points in S must also lie within S. A weighted average of points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ is a point given by the linear combination

$$\mathbf{x}^{(avg)} = \lambda_1 \mathbf{x}^{(1)} + \cdots + \lambda_N \mathbf{x}^{(N)},$$

where $0 \leq \lambda_i \leq 1$ and $\lambda_1 + \cdots + \lambda_N = 1$. You can think of the weighted average as the center of mass, where the mass of each point is given by λ_i.

In the context of binary classification, there are two important sets that are always convex:

1. In data space, the positive and negative regions are both convex. Both regions are half-spaces, and it should be visually obvious that half-spaces are convex. This implies that if inputs $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(N)}$ are all in the positive region, then any weighted average must also be in the positive region. Similarly for the negative region.

2. In weight space, the feasible region is convex. The rough mathematical argument is as follows. Each good region (the set of weights which correctly classify one data point) is convex because it's a half-space. The feasible region is the intersection of all the good regions, so it must be convex because the intersection of convex sets is convex.

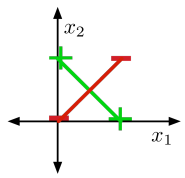

Figure 3: The XOR function is not linearly separable.

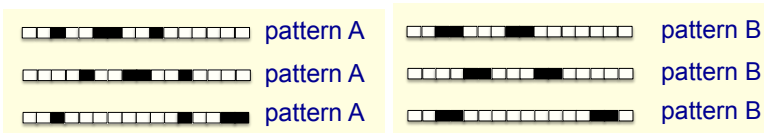

Figure 4: No linear hypothesis can separate these two patterns in all possible translations (with wrap-around).

5.2 Showing that functions aren't linearly separable

Now let's see how convexity can be used to show functions aren't linearly separable.

Example 3. *Let's return to the XOR example. Since the positive region is convex, if we draw the line segment connecting the two positive examples* $(0, 1)$ *and* $(1, 0)$*, this entire line segment must be classified as positive. Similarly, if we draw the line segment connecting the two negative examples* $(0, 0)$ *and* $(1, 1)$*, the entire line segment must be classified as negative. But these two line segments intersect at* $(0.5, 0.5)$*, which means this point must be classified as both positive and negative, which is impossible. (See Figure 3.) Therefore, XOR isn't linearly separable.*

Example 4. *Our last example was somewhat artificial. Let's now turn to a somewhat more troubling, and practically relevant, limitation of linear classifiers. Let's say we want to give a robot a vision system which can recognize objects in the world. Since the robot could be looking any given direction, it needs to be able to recognize objects regardless of their location in its visual field. I.e., it should be able to recognize a pattern in any possible translation.*

As a simplification of this situation, let's say our inputs are 16-dimensional binary vectors and we want to distinguish two patterns, A, and B (shown in Figure 4), which can be placed in any possible translation, with wrap-around. (I.e., if you shift the pattern right, then whatever falls off the right side reappears on the left.) Thus, there are 16 examples of A and 16 examples of B that our classifier needs to distinguish.

By convexity, if our classifier is to correctly classify all 16 instances of A, then it must also classify the average of all 16

instances as A. Since 4 out of the 16 values are on, the average of all instances is simply the vectors $(0.25, 0.25, \ldots, 0.25)$*. Similarly, for it to correctly classify all 16 instances of B, it must also classify their average as B. But the average is also* $(0.25, 0.25, \ldots, 0.25)$*. Since this vector can't possibly be classified as both A and B, this dataset must not be linearly separable.*

More generally, we can't expect any linear classifier to detect a pattern in all possible translations. This is a serious limitation of linear classifiers as a basis for a vision system.

5.3 Circumventing this problem by using feature representations

We just saw a negative result about linear classifiers. Let's end on a more positive note. In Lecture 2, we saw how linear regression could be made more powerful using a basis function, or feature, representation. The same trick applies to classification. Essentially, in place of $z = \mathbf{w}^T\mathbf{x} + b$, we use $z = \mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}) + b$, where $\boldsymbol{\phi}(\mathbf{x}) = (\phi_1(\mathbf{x}), \ldots, \phi_D(\mathbf{x}))$ is a function mapping input vectors to feature vectors. Let's see how we can represent XOR using carefully selected features.

Example 5. *Consider the following feature representation for XOR:*

$$\begin{aligned}\phi_1(\mathbf{x}) &= x_1\\ \phi_2(\mathbf{x}) &= x_2\\ \phi_3(\mathbf{x}) &= x_1x_2\end{aligned}$$

In this representation, our training set becomes

$\phi_1(\mathbf{x})$	$\phi_2(\mathbf{x})$	$\phi_3(\mathbf{x})$	t
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Using the same techniques as in Examples 1 and 2, we find that the following set of weights and biases correctly classifies all the training examples:

$$b = -0.5 \qquad w_1 = 1 \qquad w_2 = 1 \qquad w_3 = -2.$$

The only problem is, where do we get the features from? In this example, we just pulled them out of a hat. Unfortunately, there's no recipe for coming up with good features, which is part of what makes machine learning hard. But next week, we'll see how we can *learn* a set of features by training a multilayer neural net.