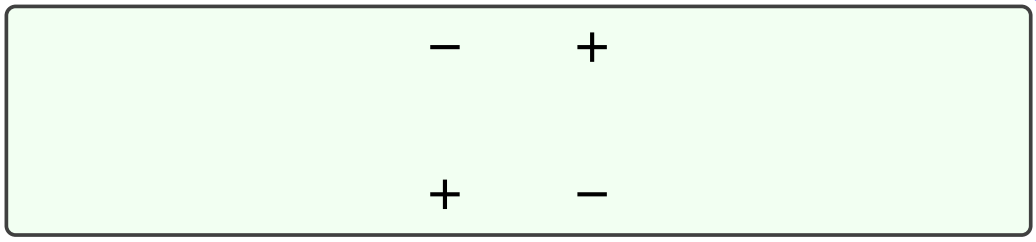
CHAPTER 4

Feature representation

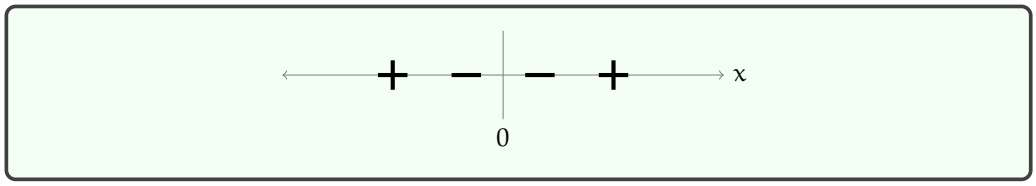
Linear classifiers are easy to work with and analyze, but they are a very restricted class of hypotheses. If we have to make a complex distinction in low dimensions, then they are unhelpful.

Our favorite illustrative example is the "exclusive or" (XOR) data set, the drosophila of machine-learning data sets:

D. Melanogaster is a species of fruit fly, used as a simple system in which to study genetics, since 1910.

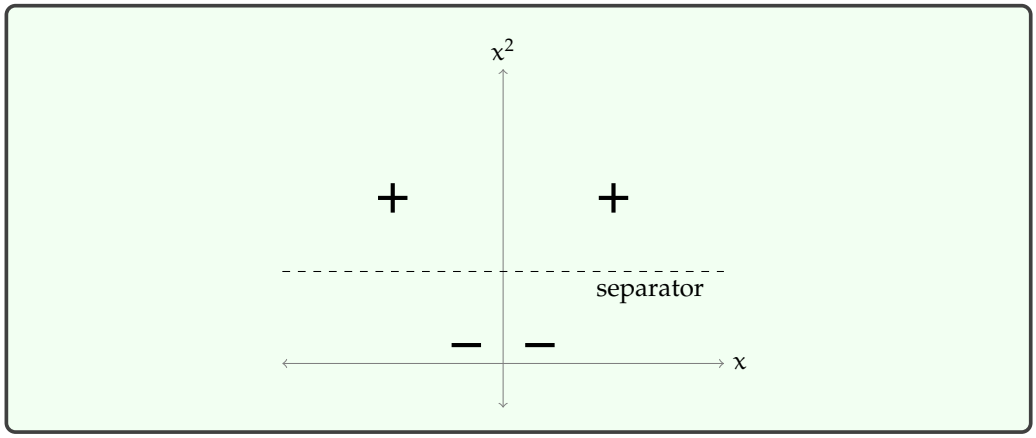


There is no linear separator for this two-dimensional dataset! But, we have a trick available: take a low-dimensional data set and move it, using a non-linear transformation into a higher-dimensional space, and look for a linear separator there. Let's look at an example data set that starts in 1-D:

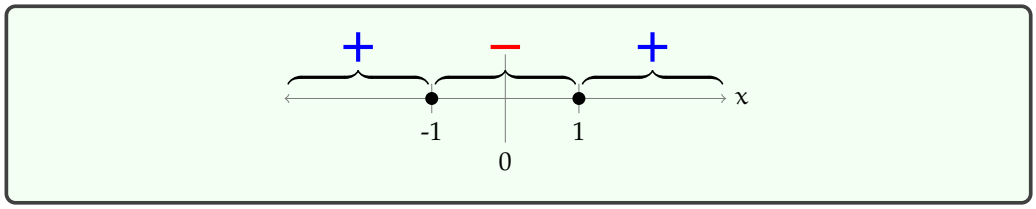


These points are not linearly separable, but consider the transformation $\phi(x) = [x, x^2]$. Putting the data in ϕ space, we see that it is now separable. There are lots of possible separators; we have just shown one of them here.

What's a linear separator for data in 1D? A point!



A linear separator in ϕ space is a nonlinear separator in the original space! Let's see how this plays out in our simple example. Consider the separator $x^2 - 1 = 0$, which labels the half-plane $x^2 - 1 > 0$ as positive. What separator does it correspond to in the original 1-D space? We have to ask the question: which x values have the property that $x^2 - 1 = 0$. The answer is $+1$ and -1, so those two points constitute our separator, back in the original space. And we can use the same reasoning to find the region of 1D space that is labeled positive by this separator.



This is a very general and widely useful strategy. It's the basis for *kernel methods*, a powerful technique that we unfortunately won't get to in this class, and can be seen as a motivation for multi-layer neural networks.

There are many different ways to construct ϕ. Some are relatively systematic and domain independent. We'll look at the *polynomial basis* in section 1 as an example of that. Others are directly related to the semantics (meaning) of the original features, and we construct them deliberately with our domain in mind. We'll explore that strategy in section 2.

1 Polynomial basis

If the features in your problem are already naturally numerical, one systematic strategy for constructing a new feature space is to use a *polynomial basis*. The idea is that, if you are using the kth-order basis (where k is a positive integer), you include a feature for every possible product of k different dimensions in your original input.

Here is a table illustrating the kth order polynomial basis for different values of k.

Order	$d = 1$	in general
0	$[1]$	$[1]$
1	$[1, x]$	$[1, x_1, \ldots, x_d]$
2	$[1, x, x^2]$	$[1, x_1, \ldots, x_d, x_1^2, x_1x_2, \ldots]$
3	$[1, x, x^2, x^3]$	$[1, x_1, \ldots, x_1^2, x_1x_2, \ldots, x_1x_2x_3, \ldots]$
$\vdots$	$\vdots$	$\vdots$

So, what if we try to solve the XOR problem using a polynomial basis as the feature transformation? We can just take our two-dimensional data and transform it into a higher-

dimensional data set, by applying ϕ. Now, we have a classification problem as usual, and we can use the perceptron algorithm to solve it.

Let's try it for $k = 2$ on our XOR problem. The feature transformation is

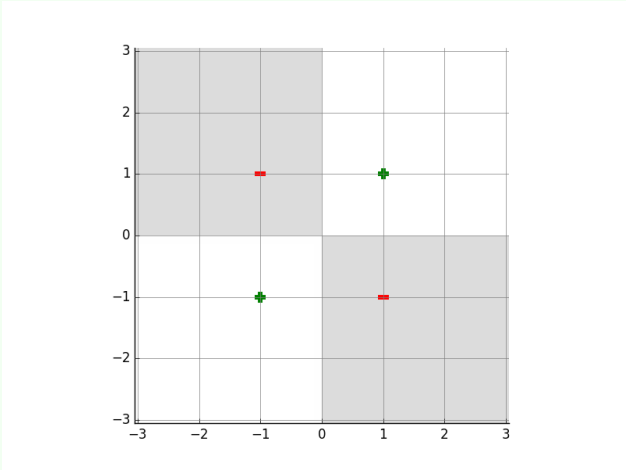
$$\phi((x_1, x_2)) = (1, x_1, x_2, x_1^2, x_1x_2, x_2^2) \ .$$

Study Question: If we use perceptron to train a classifier after performing this feature transformation, would we lose any expressive power if we let $\theta_0 = 0$ (i.e. trained without offset instead of with offset)?

After 4 iterations, perceptron finds a separator with coefficients $\theta = (0, 0, 0, 0, 4, 0)$ and $\theta_0 = 0$. This corresponds to

$$0 + 0x_1 + 0x_2 + 0x_1^2 + 4x_1x_2 + 0x_2^2 + 0 = 0$$

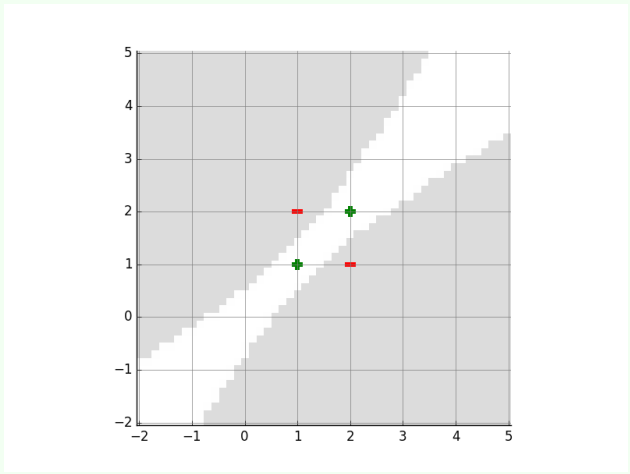
and is plotted below, with the gray shaded region classified as negative and the white region classified as positive:



Study Question: Be sure you understand why this high-dimensional hyperplane is a separator, and how it corresponds to the figure.

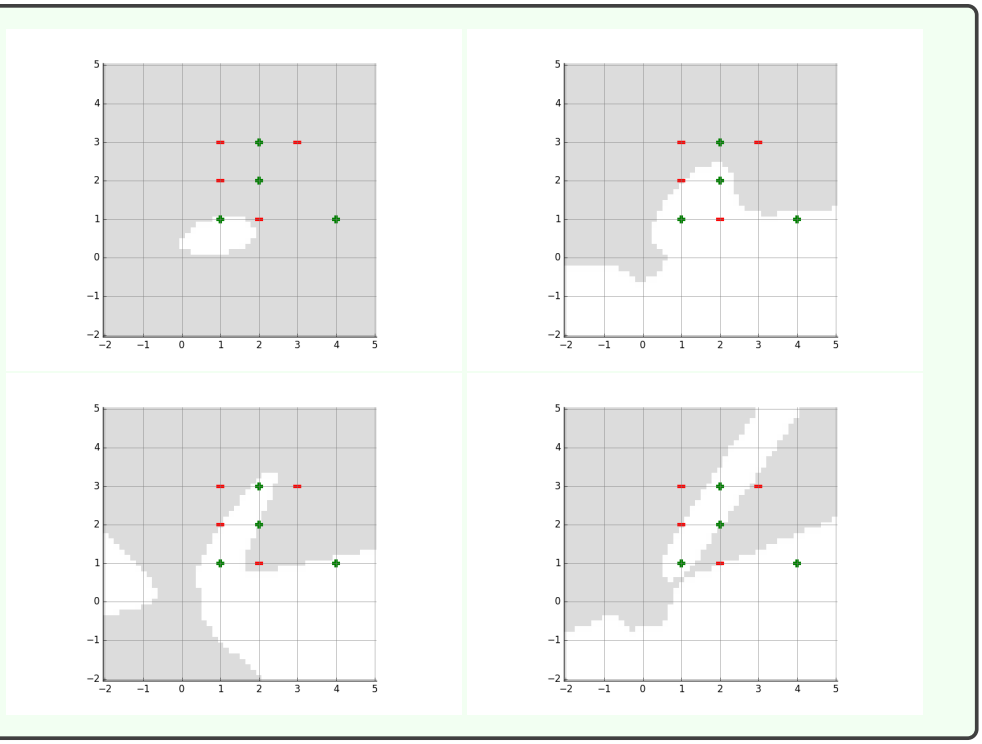
For fun, we show some more plots below. Here is the result of running perceptron on XOR, but where the data are put in a different place on the plane. After 65 mistakes (!) it arrives at these coefficients: $\theta = (1, -1, -1, -5, 11, -5)$, $\theta_0 = 1$, which generates this separator:

The jaggedness in the plotting of the separator is an artifact of a lazy lpk strategy for making these plots–the true curves are smooth.



Study Question: It takes many more iterations to solve this version. Apply knowledge of the convergence properties of the perceptron to understand why.

Here is a harder data set. After 200 iterations, we could not separate it with a second or third-order basis representation. Shown below are the results after 200 iterations for bases of order 2, 3, 4, and 5.



2 Hand-constructing features for real domains

In many machine-learning applications, we are given descriptions of the inputs with many different types of attributes, including numbers, words, and discrete features. An impor-

tant factor in the success of an ML application is the way that the features are chosen to be encoded by the human who is framing the learning problem.

2.1 Discrete features

Getting a good encoding of discrete features is particularly important. You want to create "opportunities" for the ML system to find the underlying regularities. Although there are machine-learning methods that have special mechanisms for handling discrete inputs, all the methods we consider in this class will assume the input vectors x are in $\mathbb{R}^d$. So, we have to figure out some reasonable strategies for turning discrete values into (vectors of) real numbers.

We'll start by listing some encoding strategies, and then work through some examples. Let's assume we have some feature in our raw data that can take on one of k discrete values.

- **Numeric** Assign each of these values a number, say $1.0/k, 2.0/k, \ldots, 1.0$. We might want to then do some further processing, as described in section 8.3. This is a sensible strategy *only* when the discrete values really do signify some sort of numeric quantity, so that these numerical values are meaningful.
- **Thermometer code** If your discrete values have a natural ordering, from $1, \ldots, k$, but not a natural mapping into real numbers, a good strategy is to use a vector of length k binary variables, where we convert discrete input value $0 < j \leqslant k$ into a vector in which the first j values are 1.0 and the rest are 0.0. This does not necessarily imply anything about the spacing or numerical quantities of the inputs, but does convey something about ordering.
- **Factored code** If your discrete values can sensibly be decomposed into two parts (say the "make" and "model" of a car), then it's best to treat those as two separate features, and choose an appropriate encoding of each one from this list.
- **One-hot code** If there is no obvious numeric, ordering, or factorial structure, then the best strategy is to use a vector of length k, where we convert discrete input value $0 < j \leqslant k$ into a vector in which all values are 0.0, except for the jth, which is 1.0.
- **Binary code** It might be tempting for the computer scientists among us to use some binary code, which would let us represent k values using a vector of length $\log k$. *This is a bad idea!* Decoding a binary code takes a lot of work, and by encoding your inputs this way, you'd be forcing your system to *learn* the decoding algorithm.

As an example, imagine that we want to encode blood types, which are drawn from the set $\{A+, A-, B+, B-, AB+, AB-, O+, O-\}$. There is no obvious linear numeric scaling or even ordering to this set. But there is a reasonable *factoring*, into two features: $\{A, B, AB, O\}$ and $\{+, -1\}$. And, in fact, we can reasonably factor the first group into $\{A, \text{not}A\}$, $\{B, \text{not}B\}$ So, here are two plausible encodings of the whole set:

It is sensible (according to Wikipedia!) to treat O as having neither feature A nor feature B.

- Use a 6-D vector, with two dimensions to encode each of the factors using a one-hot encoding.
- Use a 3-D vector, with one dimension for each factor, encoding its presence as 1.0 and absence as -1.0 (this is sometimes better than 0.0). In this case, $AB+$ would be $(1.0, 1.0, 1.0)$ and $O-$ would be $(-1.0, -1.0, -1.0)$.

Study Question: How would you encode $A+$ in both of these approaches?

2.2 Text

The problem of taking a text (such as a tweet or a product review, or even this document!) and encoding it as an input for a machine-learning algorithm is interesting and complicated. Much later in the class, we'll study sequential input models, where, rather than having to encode a text as a fixed-length feature vector, we feed it into a hypothesis word by word (or even character by character!).

There are some simpler encodings that work well for basic applications. One of them is the *bag of words* (BOW) model. The idea is to let d be the number of words in our vocabulary (either computed from the training set or some other body of text or dictionary). We will then make a binary vector (with values 1.0 and 0.0) of length d, where element j has value 1.0 if word j occurs in the document, and 0.0 otherwise.

2.3 Numeric values

If some feature is already encoded as a numeric value (heart rate, stock price, distance, etc.) then you should generally keep it as a numeric value. An exception might be a situation in which you know there are natural "breakpoints" in the semantics: for example, encoding someone's age in the US, you might make an explicit distinction between under and over 18 (or 21), depending on what kind of thing you are trying to predict. It might make sense to divide into discrete bins (possibly spacing them closer together for the very young) and to use a one-hot encoding for some sorts of medical situations in which we don't expect a linear (or even monotonic) relationship between age and some physiological features.

If you choose to leave a feature as numeric, it is typically useful to *scale* it, so that it tends to be in the range $[-1, +1]$. Without performing this transformation, if you have one feature with much larger values than another, it will take the learning algorithm a lot of work to find parameters that can put them on an equal basis. So, we might perform transformation $\phi(x) = \frac{x - \overline{x}}{\sigma}$, where $\overline{x}$ is the average of the $x^{(i)}$, and σ is the standard deviation of the $x^{(i)}$. The resulting feature values will have mean 0 and standard deviation 1. This transformation is sometimes called *standardizing* a variable .

Such standard variables are often known as "z-scores," for example, in the social sciences.

Then, of course, you might apply a higher-order polynomial-basis transformation to one or more groups of numeric features.

Study Question: Percy Eptron has a domain with 4 numeric input features, $(x_1, \ldots, x_4)$. He decides to use a representation of the form

$$\phi(x) = \text{PolyBasis}((x_1, x_2), 3) \frown \text{PolyBasis}((x_3, x_4), 3)$$

where $a \frown b$ means the vector a concatenated with the vector b. What is the dimension of Percy's representation? Under what assumptions about the original features is this a reasonable choice?